A STUDY ON SYSTEMS FOR FUNCTIONAL VERIFICATION AND PERFORMANCE EVALUATION IN SOFTWARE DESIGN

Kiyoshi ITOH

Doctoral Thesis

Department of Information Science
Faculty of Engineering
Kyoto University
Kyoto, Japan

1980

A STUDY ON SYSTEMS FOR FUNCTIONAL VERIFICATION AND PERFORMANCE EVALUATION IN SOFTWARE DESIGN

Kiyoshi ITOH

Doctoral Thesis

Department of Information Science
Faculty of Engineering
Kyoto University
Kyoto, Japan
1980

A Study on Systems for Functional Verification and Performance Evaluation in Software Design

by Kiyoshi ITOH

Abstract

In the early software development cycle, no powerful methodology was adopted in the requirements specification stage or design stage while many traditional tools or methods were used for testing or debugging software systems. A large number of man-hours were wasted in coding and testing software systems which were developed from erroneous requirements specifications or errorneous design.

Software Requirements Engineering enables the formalized requirements specifications for software systems to be built and to be verified as automatically as possible with the aid of computers. Main purpose is that the valid requirements specifications can be conveyed to software designers. No new design philosophy will ever improve the reliability of software systems if the requirements specifications upon which it is based are incorrect.

Systematic methodologies are necessary in the design stage of software systems, for the ambiguous, imcomplete or inconsistent design of software systems must not be conveyed to their implementation stage. Even if requirements specifications are valid, no new elaborated implementation methodology will ever produce reliable software systems if the design process proceeds incorrectly. In the design stage, the design must be performed rigourously according to design disciplines and they must be fully verified or evaluated for their function and performance.

In this research, two systems are developed for the functional verification and performance evaluation of software systems which are being designed. One of the systems is well suited for the functional verification and performance evaluation of software systems which are being designed in top-down fashion. The other system is well suited for the functional verification and performance evaluation of software systems which are being designed in their detail.

The first system is called the Interactive Modeling and Simulation System (IMSS), in which the online simulation ability and the process of top-down modeling and simulation execution are integrated. IMSS conforms well to the simulation of object software systems which are being designed in top-down fashion. As the software designer can interactively build a hierarchical simulation model corresponding to a hierarchical software system under design, there exists the opportunity of verifying the software system. As he can interactively execute the hierarchical simulation model at the same level that the corresponding software system is being designed, he can obtain performance information of the object system under design.

The second system is called the System Description and Evaluation System (SDES). SDES is applicable to the functional verification and performance evaluation for software systems which are being designed in detail. The detailed design of software systems means that the software designer determines the detailed individual behavior of software modules and intermodulle communication or interaction after he designs the whole module structure of the software system. The software designer has only to construct a simple description for verification and evaluation. The SDES verification/evaluation system executes the original design description and examines the functions and performance of the software system with the aid of the description for verification and evaluation. The design description and the

description for verification and evaluation have a complementary relationship with each other.

These two systems are useful support systems by which design errors can be deleted by the continual functional verification and performance evaluation for software systems during their design. Two systems were applied to functional verification and performance evaluation of an online software system which was being designed. IMSS has been developed on the large-scale computers FACOM M-190 and M-200 in Data Processing Center of Kyoto University. SDES has been developed on the large-scale computers HITAC M-180 in the Educational Center for Information Processing of Kyoto University and FACOM M-190 and M-200 in the Data Processing Center of Kyoto University.

Acknowledgement

I would like to express my hearty gratitude to Professor Yutaka Ohno for his guidance, constant support and encouragement during the course of this research, and also to Associate Professor Koichi Tabata for his continual support and stimulating discussion. I gratefully acknowledge Research Associates Kiyoshi Agusa and Masatoshi Kubo for stimulating discussions. I am indebted to Miss Sumali Maungpaison, Mr. Ryuichi Ban, Mr. Masahiro Inoue, Mr. Tsuyoshi Masaki, Mr. Toshinori Nagai and Mr. Shigeo Sugimoto for their discussions and cooperations.

In completing this thesis, I wish to thank to Professor Shigemichi Suzuki and Dr. Naofumi Matsumoto (Laboratory of Systems Engineering, Department of Engineering Mechanics, Faculty of Science and Technology, Sophia University) for their support and encouragement.

A Study on Systems for Functional Verification
and Performance Evaluation
in Software Design

Table of Contents

List of Figures and Tables

Figures

Tables

Chapter I
Introduction

Section I.1
Introduction

Scientific and systematic methodologies are necessary for the development of large-scale software systems in order to increase its productivity and to produce more reliable software systems. A large number of man-hours have been wasted for many years in developing software systems with unsatisfactory methodologies Software Engineering, which was started in the late 1960's, attempts to put the scentific and systematic methodologies to practical use in this field.

The development activity of software systems is divided into three stages in a broad sense, i.e., requirements specification stage, design stage and implementation stage. At the first stage, decisions are made not of how software systems execute their jobs but simply of what jobs they execute. At the second stage, the components of software systems, called modules or tasks, are enumerated and interrelationships between components, called module structure or task structure, are determined. At the third stage, software systems are built in certain programming languages based on underlying computer systems.

Fig.I.1.1 shows the early software development cycle as modified from of [Belford 1976]. In the specification stage of the

requirements for software systems, the specifications were often specified in non-technical or natural languages, so they were apt to include the ambiguity, incompleteness or inconsistency in them. In the implementation stage of software systems, the implementation was performed repeatedly with examining their functions and performance by the use of traditional test and debug tools. Specification errors were often not detected until this implementation stage due to the ambiguity, incompleteness or inconsistency of the requirements specification, often making it necessary to return to the requirements specification stage. This means that a large number of man-hours were wasted in designing, coding and testing software systems which were developed from erroneous requirements specifications. It involves enormous costs to return from the implementation stage to the requirements specification stage and to correct specification errors in the specification stage. It was also common to detect design errors in the implementation stage because the design of software systems did not faithfully follow systematic methodologies, and further was not fully verified or evaluated.

Software Requirements Engineering enables the formalized requirements specifications for software systems to be built and then verified as automatically as possible with the aid of computers. Its main purpose is that the valid requirements specifications can be conveyed to software designers. No new design philosophy will ever improve the reliability of software systems if the requirements specifications upon which it is based are incorrect. Fig.I.1.2 shows this improved software development cycle which is also modified one of [Belford 1976]. Note that in Fig.I.1.2, the feed-back line from the implementation stage to the requirements specification stage has disappeared.

Fig.I.1.3 shows an even more improved software development cycle. In Fig.I.1.3, the feed-back line from the implementation

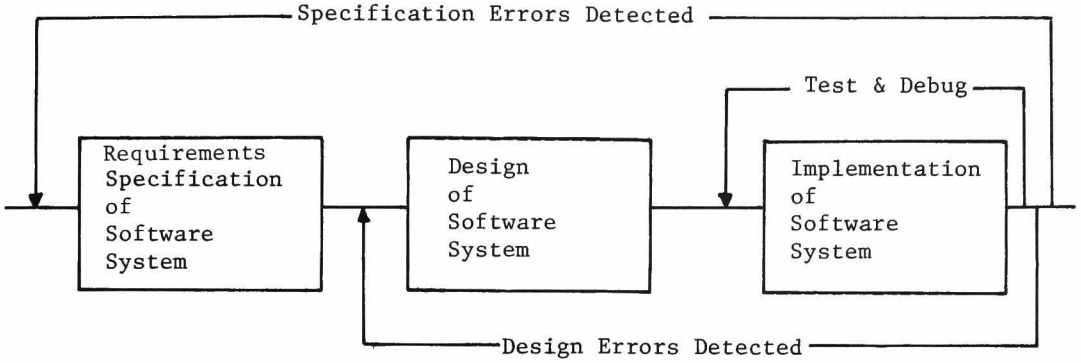


Fig.I.1.1 Early Development Cycle of Software System

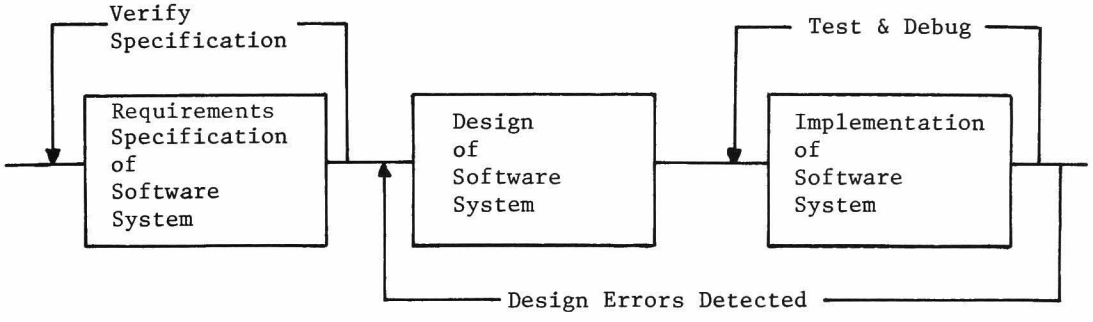


Fig.I.1.2 Development Cycle of Software System Improved by Requirements Engineering

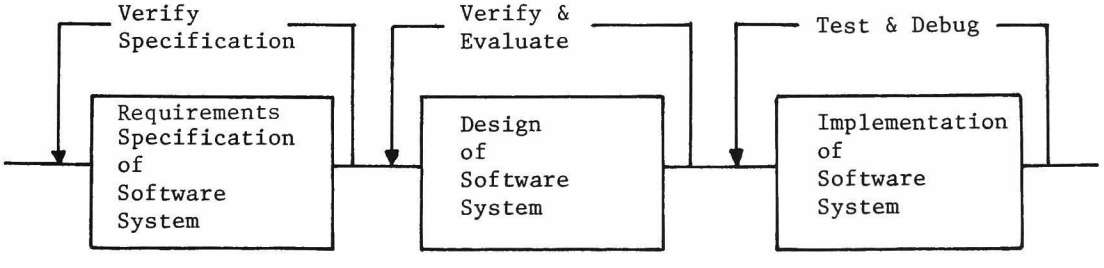


Fig.I.1.3 Development Cycle of Software System More Improved by Rigorous Design Methodology

stage to the design has disappeared, i.e., the ambiguous, incomplete or inconsistent design of software systems must not be conveyed to their implementation stage. In the design stage, the design of software systems must be performed rigorously according to design disciplines and the software systems themselves must be fully verified and evaluated for their functions as well as performance. Even if requirements specifications are valid, no new elaborated implementation methodology will ever produce reliable software systems if the design process proceeds incorrectly. At the design stage of software systems, the functional verification and performance evaluation mean an activity in which software systems under design are examined as to whether they hold required functions and required performance presented in the requirements specification.

In this research, two systems are proposed for the functional verification and performance evaluation of software systems which are under design. One of the systems is well suited for the functional verification and performance evaluation of software systems which are being designed in top-down fashion. The other system is well suitable for the functional verification and performance evaluation of software systems which are being designed in detail.

The first system is called the Interactive Modeling and Simulation System (IMSS), in which the online simulation ability and the process of top-down modeling and simulation execution are integrated. IMSS conforms well to the simulation of object software systems which are being designed in top-down fashion. As the software designer can build a hierarchical simulation model corresponding to the hierarchical software system under design, the software system can easily be verified. As he can execute the hierarchical simulation model at the same level that the corresponding software system is being designed, he can obtain performance information about the object systems under design.

The second system is called the System Description and Evaluation System (SDES). SDES is applicable to the functional verification and performance evaluation for software systems which are being designed in detail. The detailed design of software systems means that the software designer determines the detailed individual behavior of software modules and intermodulle communication or interaction after he designs the whole module structure of the software systems. A software designer has only to construct a simple description for verification and evaluation. The SDES verification/evaluation system executes the original design description and examines the functions and performance of the software system with the aid of the description written for verification and evaluation. The design description and the description for verification and evaluation have a complementary relationship with each other.

These two systems are useful support systems by which design errors can be deleted through the continual functional verification and performance evaluation for software systems on the progress of their design.

Section I.2 surveys the existing methodologies used for the requirements specification, design and implementation of software systems. Section I.3 describes new functional verification and performance evaluation systems for the desin of software systems.

Section I.2
Methodologies for Software Development

I.2.1 Methodologies for Requirements Specification

The key methodologies for the requirements specification for software systems are Structured Analysis and Design Technique (SADT)[Ross 1977], Information System Development and Optimization System(ISDOS) [Teichroew 1977], and Software Requirement Methodology(SREM) [Bell 1977].

[SADT]

In SADT, a requirements specification is defined in terms of hierachical diagrams named a SADT model. The SADT model is constructed in stepwise fashion down from higher levels, the model becoming more detailed as it moves downward. A SADT model is composed of boxes and arrows. A box corresponds to a separate activity or piece of data, and arrows represent interfaces between boxes. When it represents an activity, four arrows represent data input to it, output from it, control data for it, and mechanism implementing it. When it represents data, these four arrows represent activity generating it, activity using it, activity controlling it and mechanism storing it. Boxes and arrows are labeled in terms of a non-technical natural language. SADT makes use of forty symbols on arrows.

SADT enables requirements analysts to define a requirements specification in graphical notation, so that it may be verified and reviewed by several analysts. But, both because the graphical notation does not follow rigorous definitions and a

natural language is adopted, it is difficult to verify such requirements specifications with the aid of a computer. Further, SADT pays little attention to performance in the requirements specification stage.

[ISDOS]

In ISDOS, a computer-aided structured documentation and analysis technique is provided for functional requirements analysis for information processing systems. Functional requirements are specified in terms of a formal language named Problem Statement Language (PSL) which is based on a binary relational model. The functional requirements deal with "system input/output flow", "system structure", "data structure", "data derivation", "system size and volume", "system dynamics","system properties" and "project management".

The Problem Statement Analyzer (PSA) enters the functional requirements expressed in PSL into a data base, from which standard reports such as "data base modification reports", "reference reports", "summary reports" and "analysis reports" are produced. Analysis reports deal with apparent incompleteness or inconsistency in the requirements specification.

[SREM]

In SREM, functional requirements are specified in Requirements Specification Language (RSL) which is a flow-oriented language suited to specifying requirements for real-time information processing systems. RSL enables not only specifications to be constructed in a manner similar to that in PSL, but also enables definition of graphical structures named R-nets in which flows of messages described in Pascal are embedded. Validation points may be inserted in R-nets in order to specify performance requirements of information processing sys-

tems. The Requirements Engineering and Validation System (REVS) checks the completeness and consistency of requirements specifications and performs simulation with the aid of requirements for performance information specified at these validation points.

I.2.2 Methodologies for Software Design

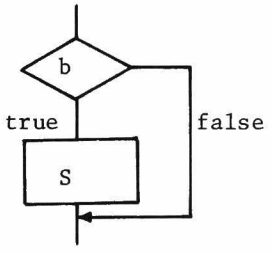
Module structure is determined at the design stage of software systems. Existing methodologies enable software designers to carry out his design process of software systems effectively using disciplined rules which prevent design errors from entering the design process. Simulation and analytical methods such as queuing analysis are frequently adopted in order to evaluate the performance of a software system under development.

The key methodologies for the design of software systems are Structured Programming [Dahl 1972], Stepwise Refinement [Wirth 197la], Top-Down Design [Mills 1971], Modularization by Information Hiding [Parnas 1971], Modularization by Data Abstraction [Liskov 1974], Composite Design [Myers 1975], LOGOS [Rose 1972], and the Design and Evaluation System (DES) [Graham 1973].

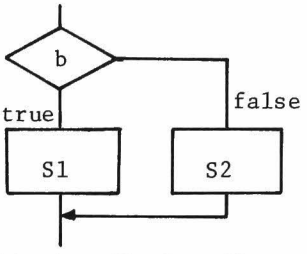
[Structured Programming]

The premise of the Structured Programming is to use a small set of control structures, i.e., sequence, selection and repetition shown in Fig.I.2.1. These are structures with one entry and one exit. A program is then built by nesting these structures inside each other. This method restricts the number of connections between program components and thereby improves the com-

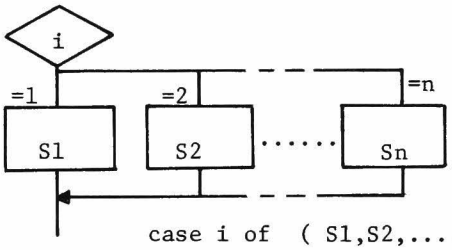
(1) Selection Structure



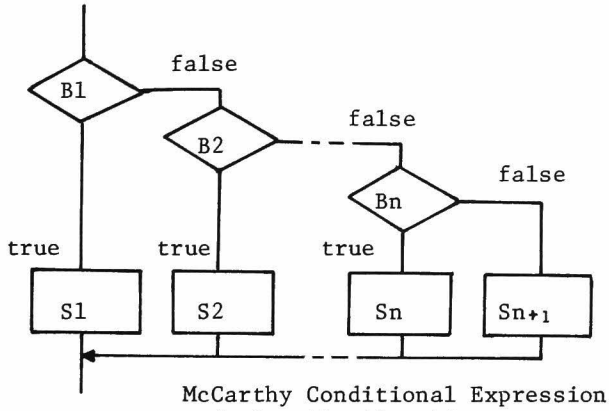
if b then S;



if b then S1 else S2;



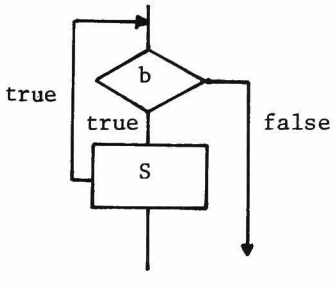
case i of (S1,S2,.....,Sn) ;



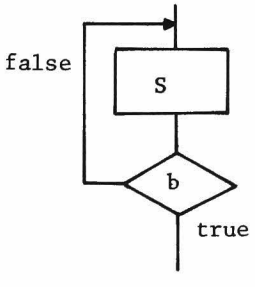
McCarthy Conditional Expression

(B1-> A1, B2-> A2, ..., Bn-> An, Bn_{+1}-> An_{+1});

(2) Repetition Structure



while b do S;



repeat S until b;

Fig.I.2.1 Set of Control Structures in Structured Programming

prehensibility and reliability of a program, so software design can be performed smoothly.

[Stepwise Refinement]

This method develops a program gradually and hierarchically in a sequence of refinement steps. At each step, one or more instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement continues until all instructions are expressed in terms of an underlying programming language. Every refinement step requires the refinement of related data. This method enables decisions for refining of trivial parts of the program to be defered and decisions for the refining its more important parts to be carried out.

[Top-Down Design]

This method develops large-scale software systems in an evolving tree structure of nested program modules, with no control branching between modules except for module calls defined in the tree structure. This leads to effective software design by limiting the size and complexity of modules.

Structured Programming provides principles for the control structure of each software module, Stepwise Refinement provides principles for the refinement of software modules, and Top-down Design provides principles for the whole module structure of software systems. The combination of these three methodologies enables software designers to produce reliable software systems smoothly with comprehensive structure.

[Modularization by Information Hiding]

This method controls the distribution of information in software systems,i.e., it inhibits a module to access to information which is irrelevant to the module.

[Modularization by Data Abstraction]

This method modularizes software systems by defining an abstract data type called an operation cluster which is a class of objects characterized by the operations which may be performed on them. The software designer need only be aware of the behavior of operation clusters, and irrelevant details about how the data is represented in storage and how the operations are implemented are hidden from him.

[Composite Design]

In order to reduce the complexity of software systems by dividing them into functional modules and to create complex systems from simple, independent and reusable modules, Composite Design methodology provides the analysis method based on "module strength" and "module coupling". The module strength is the measure on the relationships of elements in a module. The types of the module strength are "coincidental strength", "logical strength", "classical strength", "procedural strength", "communicational strength", "informational strength", and "functional strength". The module coupling is the measure on the interrelationships between modules. The types of the module coupling are "content coupling", "common coupling", "external coupling", "control coupling", "stump coupling" and "data coupling". Composite Design methodology produces reliable software systems which have high module strength and low module coupling.

[LOGOS]

In order to reduce problems of complexity -- intermodule communication, software/hardware interface conflicts and mishandling of real and apparent concurrency within the hardware/software system, the designers describe a control flow graph and a data flow graph which are analyzed for the detection of their incompleteness or inconsistency and for the decisions of the trade-offs of functions between the software and hardware systems functions.

[DES]

In order to integrate the performance evaluation of a software system with its design, a high level programming language, which is similar to a simulation language, is used to describe the software system. This description is inputted directly to the performance analysis and simulation routines.

I.2.3 Methodologies for Software Implementation

[Language for Software Implementation]

At the implementation stage of software systems, many high level programming lanuages are provided which are especially suited to the methodologies of Structured Programming and Modularization by Data Abstraction. An arbitrary programming language is suited to Top-Down Design or Stepwise Refinement.

Languages named Bliss [Wulf 1971b], Pearl [Snowden 1972], Pascal[Wirth 1971] are used for Structured Programming. They

enable a programmer to build his program clearly in terms of IF THEN (ELSE) statement or CASE statement for selection structure REPEAT UNTIL statement or DO WHILE statement for repetition structure.

When implementing software systems with Top-Down Design, procedures, subroutines or macro instructions are assigned to functions which have been enumerated in the design stage and then a sequence of calling statements is described for them. This proess is also repeated to each of them. The resultant structure has a tree structure or hierarchical structure of modules. A program being developed in top-down fashion can be executed and tested by the assigning of program stubs to yet-to-be-designed/implemented modules.

Simula 67 [Ichbiah 1972], CLU [Liskov 1974] and ALPHARD [Wulf 1976] are languages suited to the methodologies of Modularization by Data Abstraction.

PL/I is the programming language most widely used for implementing of software systems.

[Verification and Evaluation of Software]

There are many long-established tools for verifying the functions of software systems.

As a tool for checking program structure, a tool for program graph generation, well formation checks and loop termination checks is provided [Ramamoorthy 1966]. A Program graph is a directed graph with nodes representing statements and arcs representing program control flow.

In order to detect the errors in sequences of certain specified

events, a tool extracts those automatically from program code [Howden 1973].

In order to monitor the run-time behavior of a program, codes for checking the bounds of variables, for tracing variables, for recording the frequency of referencing to certain parts of programs and for tracing execution paths are inserted in program codes.

A tool is constructed to generate test cases which satisfy that every executable statement and every possible outcome of each branch statement can be execised at least once.

A tool based on the Mathematical Theory of Computation checks assertions which are invariant conditions for the ranges of values of variables or the relations of several variables [Good 1975].

Simulation techniques are frequently used to evaluate the performance of software systems and this simulation is often performed in parallel with the implementation of software systems.

Section I.3
Two Systems for Functional Verification and Performance Evaluation of Software in its Design

In this research, two systems are proposed for the functional verification and performance evaluation of software systems during their design. One of the systems is well suited for application to the functional verification and performance evaluation of software systems which are being designed in top-down fashion. The other system is well suitable for the functional verification and performance evaluation of software systems which are being designed in detail.

I.3.1 General Concept of Interactive Modeling and Simulation System

The first system is called the Interactive Modeling and Simulation System (IMSS). IMSS allows software designers to build hierarchical simulation models in top-down fashion and then to execute the hierarchical simulation models at the arbitrary level of the modeling process in order to verify them or obtain their statistical properties. In IMSS, the process of top-down modeling and simulation execution at this arbitrary level is called "top-down modeling and simulation execution process".

The top-down modeling and simulation execution process conforms

well to the simulation of software systems which are being designed in top-down fashion. First, the software designer builds a hierarchicalsimulation model of the object software system at the same level that the object software system is being hierarchically designed. He can execute the simulation model at the arbitrary level of the modeling process, i.e., at the arbitrary level of designing process of the object system, in order to verify the model or obtain its statistical properties. The software designer can verify the functions of a software system and evaluate its performance by building a simulation model for it and then executing it at the same level of designing the software system. IMSS is a useful support system for the top-down design of software systems.

In the left half of Fig.I.3.1, the top-down design is shown in terms of a hierarchical structure of a software system. Components of this hierarchical structure of a system are called modules, e.g., M_{01}, M_{11},....., M_{nl} in Fig.I.3.1. The hierarchical structure of a software system is represented in an evolving tree structure, e.g., emanating to level "N" in Fig.I.3.1. In a software system being developed in top-down fashion, there is no control branching between modules except for module calls defined in its tree structure. When implementing a software system which is designed in this way, the hierarchical structure of a software system is represented as a nesting structure of subroutine, procedure or macro instruction calls.

The advantage of top-down development exists in executing a software system under development in top-down fashion using "program stubs", e.g., PS in Fig.I.3.1. In this case, already-designed modules are fully described in a certain programming language, and yet-to-be-designed modules are replaced to program stubs. These stubs are also described in the same language but contains only main control sequence and interfaces to its upper module. Using program stubs to execute a software

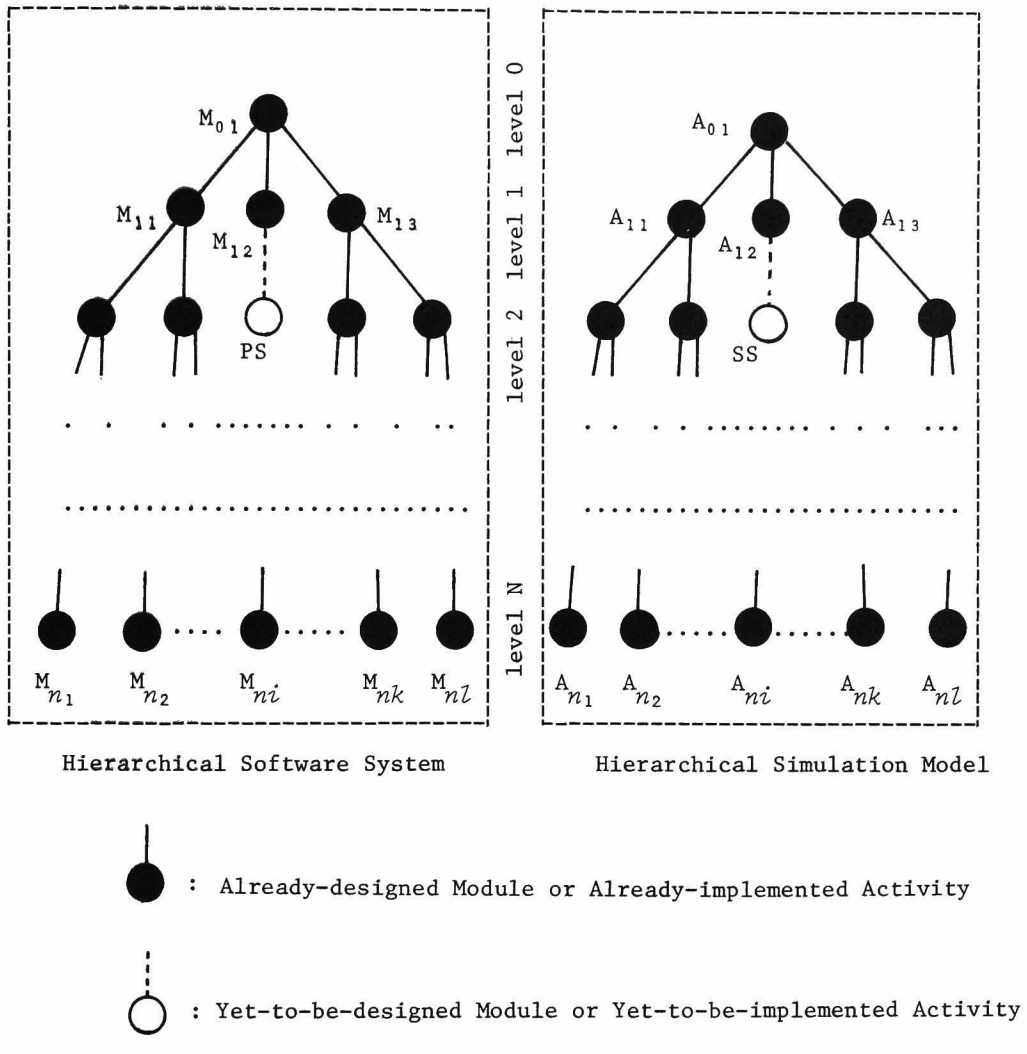


Fig.I.3.1 Correspondence between Hierarchcal Software System and its Hierarchical Simulation Model

system provides the software designer with assurance about the correctness of the design of the software system at that level especially concering already-desined modules.

In order to obtain the assurance about the valid performance of a software system being developed in top-down fashion, a hierarchical simulation model is built at a level matching the development of the hierachical software system as shown in the right half of Fig.I.3.1. Components of this hierarchical structure of a simulation model are named "activities", e.g., A_{01}, A_{11},....., A_{nl} in Fig.I.3.1, which correspond to M_{01},M_{11},....., M_{nl}, respectively. In this type of simulation model, there is no control branching between activities except for those activity calls defined in its tree structure. A hierarchical simulation model is described in a special simulation programming language named IMSS language and built as a nesting structure of activity calls. Flows of transactions are described for each activity. These transactions are viewed as the entities to be processed by the object software system. In their flows, transactions make use of simulation entities such as facilities, storages and queues, which are considered to be components of the object software system, and they expend certain time intervals.

A software designer can build a hierarchical simulation model as he reviews the design of the object software system. This increases his chances of finding design errors during design review.

A software designer can execute the simulation at the arbitrary level of his modeling process using "simulation stubs". Already-designed modules correspond to already-implemented activities. Yet-to-be-designed modules corresponds to yet-to-be-implemented activities. The yet-to-be-implemented activities are replaced by simulation stubs, in which main flows of transactions and

communication to upper activities are described. Executing a simulation model by using simulation stubs provides the software designer with assurance about valid performance of the object software system being developed.

I.3.2 General Concept of System Description and Evaluation System

The second system is called the System Description and Evaluation System (SDES). SDES is applicable to the functional verification and performance evaluation of a software system being designed in detail. The detailed design of a software sytem involves determining the detailed individual behavior of software modules and intermodule communication or interaction after the overall structure of the software system has been designed. Especially in the case where a software system consists of more than one concurrent processing module, the software system must be designed in enough validity that time-dependent misbehavior of software modules and erroneous intermodule communication can be avoided. In addition, the software system must be designed with the assurance about valid performance. SDES provides a software designer with a functional verification and performance evaluation tool for a software system whose detail is being designed before its implementation.

In SDES, each of concurrent processing modules is called a concurrent process. The detailed design of a software system begins with the description of all the communication between concurrent processes and of all the main control structure in each process. The original design is interactively refined in step-wise fashion using SDES to the continual functional verifi-

cation and performance evaluation.

In order to perform this functional verification and performance evaluation of a software system being designed in detail, the software designer may construct a simple description for verfication and evaluation. This description has a complementary relationship with the design description and it is concerned with the behavior of entities to be processed in the software system.

There are two complementary types of entities in any software system. The first type is a "processing entity", and the second type is an entity to be processed, named a "processed entity". Processing entities are concurrent processes. A software system is designed and implemented from the viewpoint of processing entities. Processed entities are messages, input data, or transactions. In SDES, the description for verification and evaluation is constructed from the viewpoint of processed entities. Processed entities are named "traversers", a name which derives from processed entities traversing the design description of a software system to verify its functions and evaluate its performance. A software designer constructs two complementary descriptions for a software system.

When a software designer constructs a description for verification and evaluation, named "traverser description", he reviews the design description. He may find design errors during this design review.

The traverser description is the description about messages, input data or transactions which are actually processed by the software system. This makes the traverser description well suited for evaluating the performance of the software system.

The design description is shown in the upper left half of Fig.I. 3.2, where P_1, P_2, ..., P_n are processes whose behaviors are

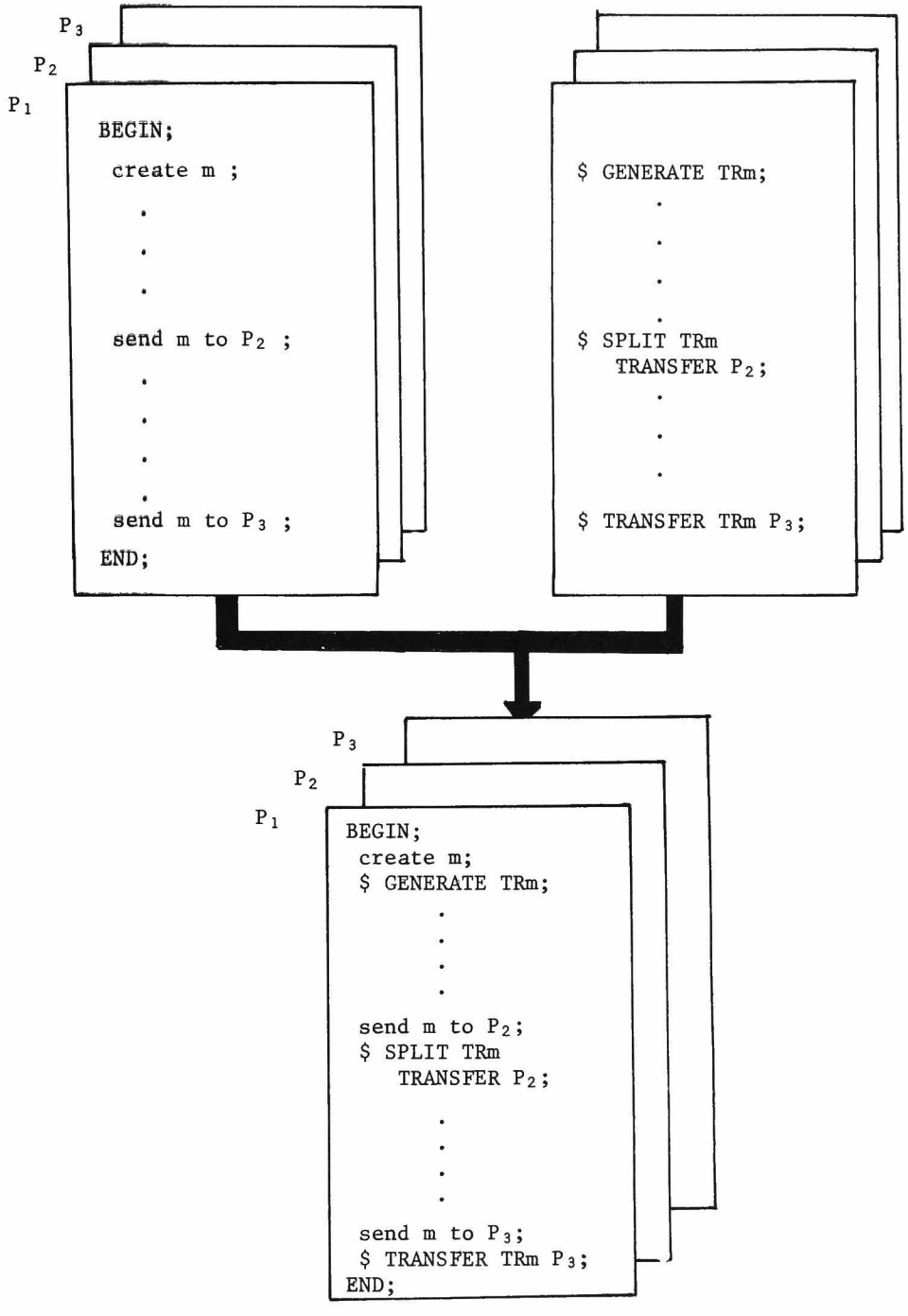


Fig.I.3.2 Verification and Evaluation of Software Design by SDES

written in PL/I in SDES. They are written in English language flavor for the sake of readability in Fig.I.3.2. The design description begins with the description of the main control structures of behavior of concurrent processes and interprocess communication. This description is refined as the design proceeds.

Traverser description is shown in the upper right half of Fig.I.3.2, where the behavior of traversers is described at each process. Traverser description is the description about the paths of the corresponding traversers in a software system, the interactions between processed entities and concurrent processes, resources in a software system, etc.. The software designer constructs a traverser description in Traverser Language and merges it with the design description, as shown in the lower half of Fig.I.3.2.

The merged description is executed by the SDES verification/evaluation system. The SDES verification/evaluation system executes PL/I statements of the description, verifies the design of a software system and evaluates it with the aid of the traverser description.

Chapter II
Interactive Modeling and Simulation System: IMSS

Section II.1
Introduction

Simulation is usually characterized as a three-mode activity. First, we build a model of the object system. Then, the model is tested for validation. Finally, the completed model is executed and we derive conclusions about the behavior of the object system being studied.

Ordinarily, these three modes must be repeated before the desired results are obtained. The reason is that the logical strucuture of the object system and the interrelationship of components of the system are so complicated as to preclude perfect modeling by forethought. Moreover, the object system likely includes a substantial number of concurrent events. This makes it exceedingly difficult to build accurate models by coventional methods. Software engineers and designers are in great need of a method or facility by which system simulation can be made accurate from the beginning.

One solution includes the online interactive use of digital computers. "Online", here, means that the user interacts with simulation tools during all the simulation process. In the modeling

mode, the user builds a model by using the interactive modeling facility. In the testing or execution mode, the user monitors the behavior of the model or submodels. If simulation execution is found to be proceeding erroneously, the user may immediately discontinue the execution, and then modify the model interactively. The user can repeat these modes until he understands the real behavior of the object system clearly.

The concept of "top-down modeling and simulation execution" can further enhance the ability of on-line simulation. The user may build a hierarchical simulation model, level by level, using an online interactive modeling facility. Further, he use this same facility to execute his hierarchical simulation model at the arbitrary level of his modeling process in order to verify the model or obtain its statistical properties. The concept of "top-down modeling and simulation execution" is well suited to the simulation of software systems being developed in top-down fashion. The software designer can perform this simulation interactively by building a hierarchical simulation model of the object software system matching the level of the object software system being developed in top-down fashion.

Section II.2
Basic Concept of IMSS :
Top-Down Modeling and Simulation Execution

II.2.1 Historical References for Online Simulation

This section surveys several online simulation systems. Ordinary simulation systems are divided into two groups, i.e., transaction-type simulation system such as General Purpose Systems Simulator(GPSS) [IBM] and event-type simulation system such as SIMSCRIPT [Markowitz 1963]. Online simulation system are also subgrouped.

In the U.S.A., at least three institutions have made extensive use of online simulation [Emshoff 1970]. One of the most advanced groups in the application of simulation is a group at the Norden Division of United Aircraft. Here, the GPSS/360 was modified to permit larger models, data libraries, and an interactive user devices.

The major improvements they realized are as follows.

1) Models of any size may be run by using direct access devices to store sections of the model.
2) The models can interact with data banks stored on direct access devices.
3) The model output can be presented to the user on a display unit as well as on a conventional printer.
4) Model generation and debugging have been improved by using the display unit.

They apparently feel that such online use is justified, not only

for debugging but also for analysis.

MITRE adapted GPSS to an online system [Ziegler 1968]. This system permits a GPSS run to be monitored while the run is in progress. The online monitor provides the ability to select and display statitics related to the GPSS entities.

M.I.T. explored the use of online simulation and developed a language called OPS(Online Programming System)-3 especially for this purpose [Greenberger 1965 1966 1967]. OPS-3 is an interactive system designed for general use in time-sharing environments. OPS-3 language is a event-type simulation language, and is more similar to SIMSCRIPT than GPSS. OPS-3 includes online capabilities for building models and running simulations. Simulation activities are scheduled, canceled, or rescheduled dynamically on an AGENDA either at a specified time or when a prescribed condition is met. The AGENDA is a time-ordered list of conditionally and uncondionally scheduled activities. The user may inspect the AGENDA or some index of performance without stopping the simulation. He can also interrupt the run with unprogrammed spections and alternations roll the simulation back to any earlier state that has been preserved.

In Japan, the Online Simulation System (OLSS-1) was implemented by the Department of Administrative Engineering of Seikei University [Yoshizawa 1975]. This is a GPSS-type online system simulator which makes it possible to build a model and execute simulation, achieving man-machine cooperation through a graphic display unit. Its operation is composed of four modes, i.e., system control mode, model building mode, execution mode, and information mode. There are three methods for controlling the execution, i.e., execution at clock update, at a specified clock value and after specified number of transactions. After any interruption of these types of execution, the requested information may be displayed. The state of interrupted model

is preserved. This preserved information is utilized when rolling the simulation back to earlier execution points.

II.2.2 Top-Down Modeling and Simulation Execution

In addition to the interactive modeling and monitoring facilities found in most online simulation systems, these systems should have the capability to integrate the three modes, i.e., modeling mode, testing mode and execution mode. Top-down process is introduced into the simulation activity for this purpose. The top-down process allows us not only to build a hierachical simulation model using "actentity", "activity", etc., but also to test and execute the hierarchical simulation model using simulation stub" at any stage while building it. This is called the "top-down modeling and simulation execution process".

The top-down modeling and simulation execution process is well suited for the simulation of software systems being developed in top-down fashion. The software designer can build his hierarchical simulation model by mapping already-designed software modules and yet-to-be-designed software module to activities and simulation stubs, respectively.

The first version of our online simulation system was the Graphical Modeling and Simulation System(GMSS) developed on two mini-computers conected to a graphic display unit [Tabata 1975] [Ohno 1976] [Kubo 1975 1976] [Itoh 1975 1976 1977a]. In order to open the facilities of GMSS to more users in various application fields in a TSS environment, we have developed the Interactive Modeling and Simulation System(IMSS) as the second

version of our online simulation system [Itoh 1978b]. IMSS is an interactive simulation system which allows top-down modeling and simulation execution with a character display/teletypewriter terminal on the TSS of the large-scale computers FACOM M-190 and M-200 (OS IV/F4) in the Data-processing Center of Kyoto University.

IMSS is a general purpose and transaction-oriented online simulation system. IMSS has two types of modeling "language", i.e., a textual language named IMSS language and a pictographic "language" named IMSS pictography. The two have straightforward correspondence with each other. The pictographic form of a simulation model allows us to detect structural errors in the model quite easily.

IMSS language is based on the Structured Programming technique which is well fit for the top-down modeling and simulation. IMSS language makes it easy to build a hierarchical model which is due to that IMSS language has not GOTO statement but several "structure" and "activity" statements instead. A simulation model is described as a set of activities in top-down fashion. In addition to the usual simulation entities such as facility, storage and queue, we introduce an "actentity" which is an abstract entity. This term refers to a model of a portion of a system whose action or activity is implemented in the next lower level of the top-down modeling process. The term "common actentity", abbreviated as "comact", is also introduced.

At the arbitrary stage in building a hierarchical simulation model, the model can be executed for dynamic testing by using "simulation stub". Simulation stub simulates the presence of yet-to-be-implemented actentity. The user can implement the simulation stub in simpler or more sophisticated fashion than "activity".

IMSS allows us to use a TSS terminal to monitor or interact with the execution process of a simulation model. IMSS has five interactive execution modes, i.e., Step-by-Step mode, Clock-by-Clock mode, Run mode, Condition mode and Roll-Back mode. After each mode of the simulation execution is completed, the state of the simulation model is outputted in numerical form or line graph form following a time axis of statistics on simulation entities. We may obtain several statistics by specifying any one of the five execution modes alternately during simulation execution. These interactive execution modes and this reporting facility enables us to debug the simulation model and to recognize the properties of the simulation model easily and effectively posible.

The IMSS Command System accepts IMSS user commands and switches modes during simulation. It accepts subcommands and performs corresponding tasks in each mode.

Section II.3
Tool and Principle in Modeling Mode

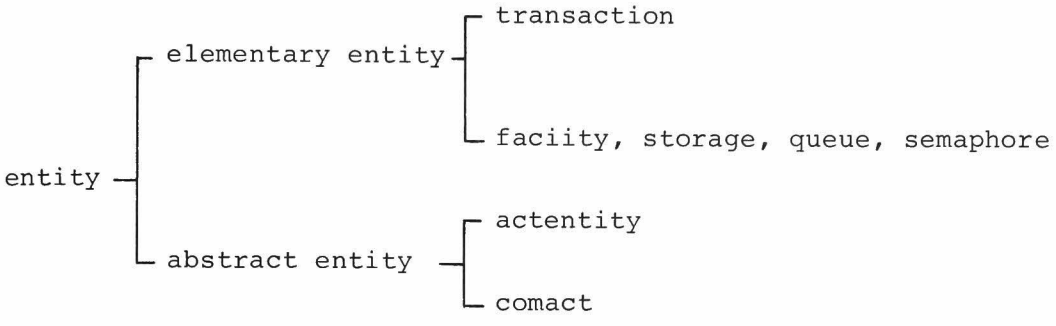
II.3.1 IMSS Language

In order to simulate any system, we need to build a model of the system. By a model of a system we mean a collection of related entities, each of which characterized by attributes that may themselves be related. An entity is an object to be simulated. The concept of event and activity is especially important when building a model of a system. An event signifies a change in state of an entity. The simulation of the system proceeds along a sequence of events ordered on time. An activity is a collection of operations that transform the states of entities.

IMSS language is a simulation language suitable for the top-down modeling and simulation execution. In IMSS, top-down process is able to be carried out not only in modeling but also in simulation execution. For this purpose, the concept of "actentity" and "simulation stub" is intoduced.

[1] Simulation Entity

In IMSS, we prepare "transaction", "facility", "storage", "queue" and "semaphore" as elementary entities. We also prepare "actentity" and "comact" which are special abstract entities in addition to elementary entities. These are listed as follows.



An entity called transaction acts on the other types of entities which include facilities, storages, actentities and so on. A transaction represents the unit of traffic which is moved through a model of a system. It enters the model, and then it acts on other entities or it is affected by various operations. Finally, it goes out of the model.

A facility entity represents a time-shared equipment. The number of transactions which can simultaneously use a facility is called the capacity of the facility. Each facility has its own service mode that means the service order for incoming the transaction. There are three types of service modes, i.e., FIFO (First-In First-Out), LIFO(Last-In Last-Out) and RAND(RANDom), each of which is specified in the declaration of a facility.

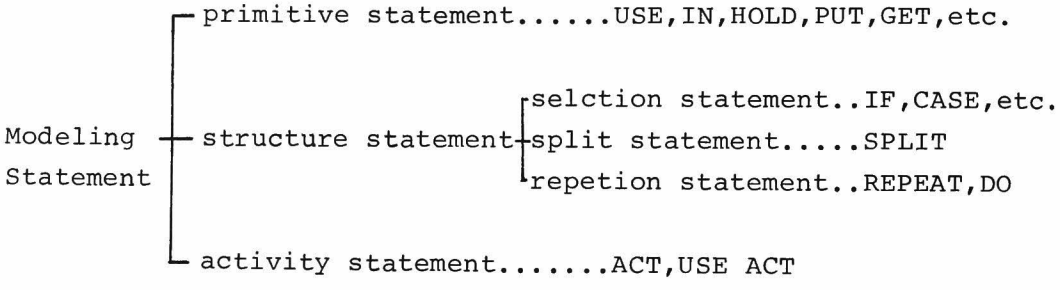
A storage entity represents a space-shared equipment whose space is required by a transaction. The space capacity is assigned in the declaration of the storage. Each storage also has its own service modes. There are four types of modes, which are FIFO, LIFO, RAND, SFO(Smallest-First-Out).

A queue entity represents an ordered set of transactions that are waiting the process of the statement just next to the queue statement corresponding to the queue entity.

A semaphore entity represents an equipment which has binary states, on or off, to which transaction may refer. Transaction can set or reset a semaphore which may be used to synchronize two or more sequences of events.

[2] Modeling Statement and ACTIVITY

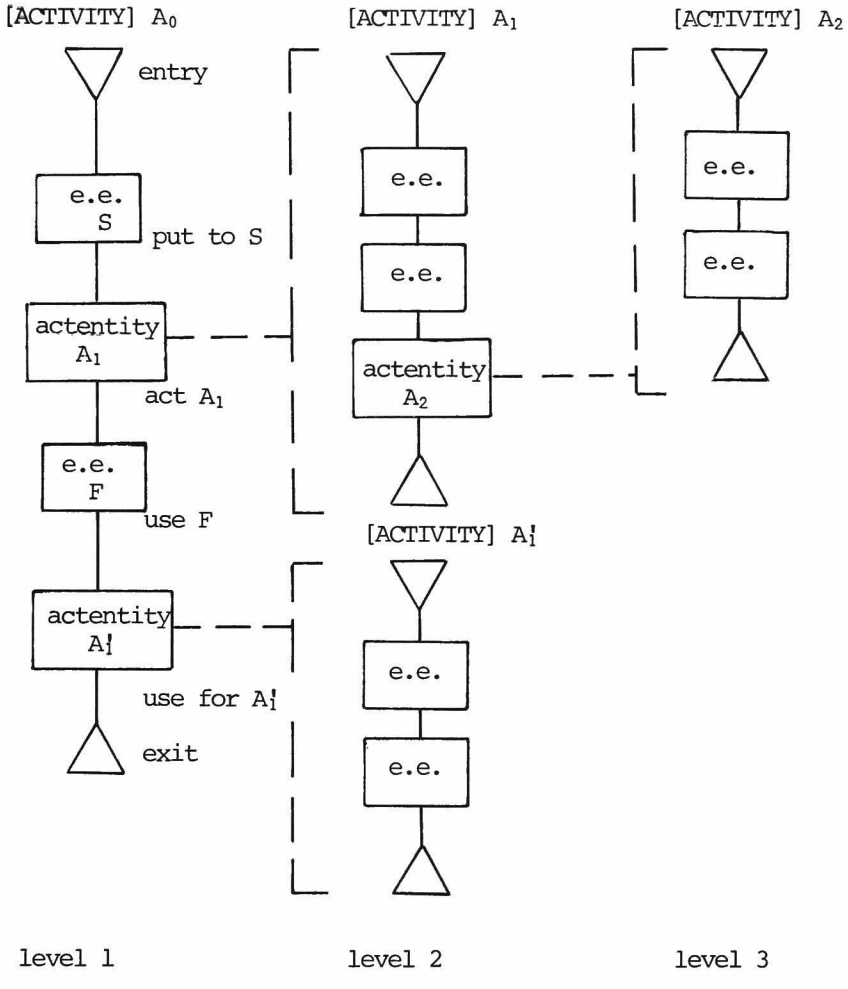
There are three types of modeling statements(operattions), primitive statements, structure statement and activity statements. Structure statements are "selection" statements, "split" statements and "repetition" statements with a single entry and a single exit, and they control the flow of transactions.



An ACTIVITY defined in IMSS language gives a set of related activities where IMSS statements(operations) change the states of IMSS entities.

[3] Actentity and Activity Statement

We explain the meaning of the actentity and the activity statement. The word "actentity" is derived from "activity entity". An actentity is an abstract entity which means a model of a certain portion of the system. An activity statement acts on a transaction, and simulates the behavior of the portion of the system corresponding to the actentity. Action of the actentity (specification) - what it does - must be fully defined, but it

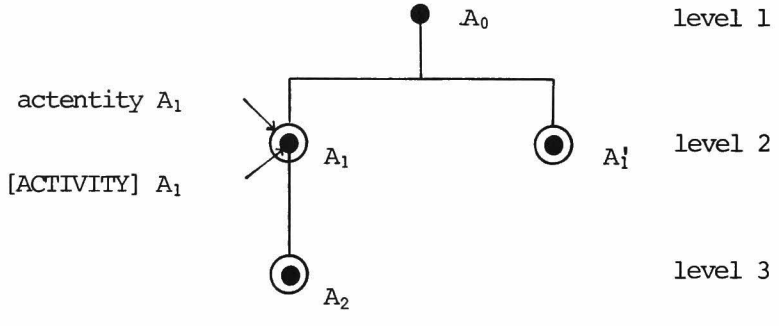


e.e. : elementary entity.

Fig.II.3.1 Levels of Modeling Stages and the Relation between Actentity and ACTIVITY

is not always necessary to know how to implement it at this point. The action or activity will be implemented in the definition of ACTIVITY in the next lower level of modeling mode. Actentity and its corresponding ACTIVITY have the identical name.

Fig.II.3.1 shows the relation between actentity and corresponding ACTIVITY in the lower level. The concept of actentity may be well suitable for top-down modeling. In order to make the structure of a model in IMSS language clear, we may give a tree structure of the system on the basis of the relation between actentity and corresponding ACTIVITY. Fig.II.3.2 shows the tree structure corresponding to Fig.II.3.1 .



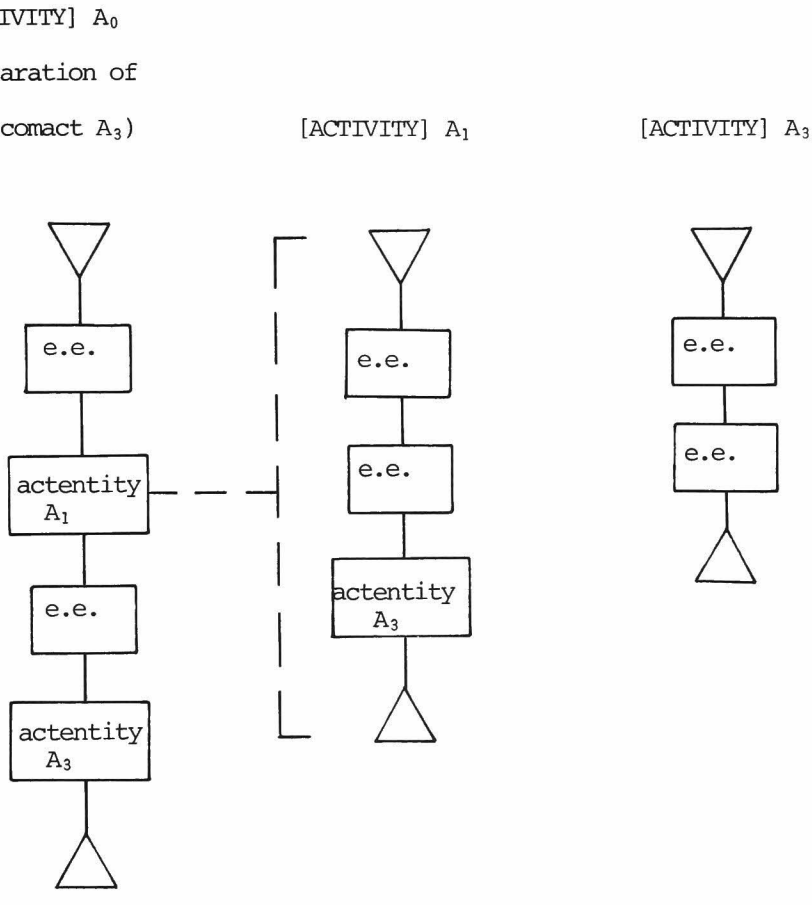
● : [ACTIVITY]

◉ : invocation for actentity (outer circle) and its corresponding [ACTIVITY] (inner black circle).

Fig.II.3.2 Tree Structure of Fig.II.3.2

[4] Comact and Model Structure

We explain the meaning of the "comact" the word of which is derived from "common actentity". A comact is a special actentity, the concept of which allows more than one different ACTIVITY to share a common actentity, and allows more than one access to one



e.e. : elementary entity.

Fig.II.3.3 Example of Comact

actentity in an ACTIVITY. Action of a comact is also implemented in the definition of ACTIVITY like an ordinary actentity. The method of access to a comact is the same as that of access to an ordinary actentity as if it were declared as the ordinary actentity in the accessing ACTIVITY.

A comact has to be declared in the definition of the root ACTIVITY of any one subtree that includes all the ACTIVITY's sharing the comact. Usually, we choose the root of the smallest subtree among such subtrees. Conversely, once a comact is declared in the definition of an ACTIVITY, it may be used in any ACTIVITY included in the suburee the root of which is the ACTIVITY with its declaration.(See Fig.II.3.3 and Fig.II.3.4.)

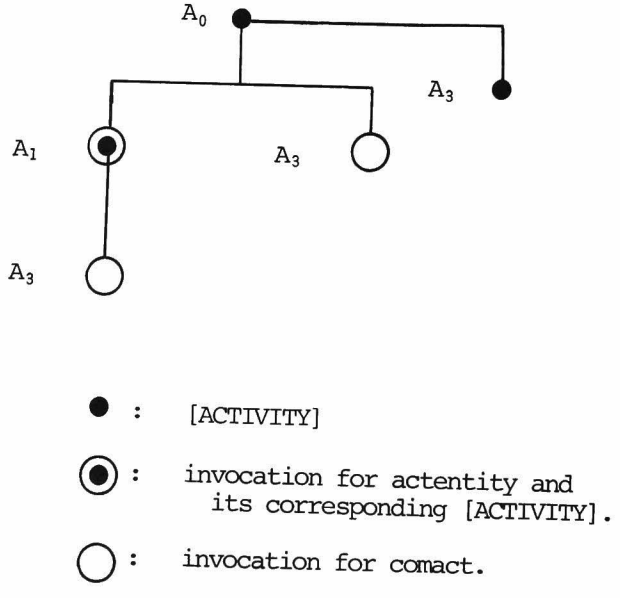


Fig.II.3.4 Tree Structure of Fig.II.3.3

[5] Function

User-defined function as well as the standard function may be called in any place of the model program. Namely, user-defined function is regarded as if it were declared in the model declaration part. As for the tree structure of the model, it may be on the horizontal branch of the root of whole tree of the model.

Standard functions may be used without declaration. UNIFORM(b,t) denotes a random integer number between b-t and b+t with uniform distribution. EXPO(m) benotes a random integer number with exponential distribution of mean m.

[6] Variable

There are three kinds of variables, a private variable, a common variable and a system variable. The type of these variable is integer.

A private variable represents one of attributes of a transaction, and its value is carried with the transaction. Transactions have their respective values for the same private variable. A transaction may have none or more private variable. A transaction can not refer to the private variable of any other transaction.

A common variable represents a state of an ACTIVITY in which it is declared. In other words, this variable is attached to the ACTIVITY. This variable may be shared by the transactions which pass the ACTIVITY.

A system variable represents one of specified attributes of an elementary entity. In the case of a facility, for example, it indicates the number of transactions which currently use the facility. It is used without declaration. All transactions in

the scope of the entity can refer to it.

[7] Array of Entity and Variable

An entity array and a variable array are permitted. An array represents a collection of variables or entities which have identical properties such as types of variables or entities, and capacities of entities.

A subscripted entity or variable is used for referring to a component of the array. Common variables, private variables or constants may be used as subscripts.

The function of the subscriped entities or variables is nearly similar to that of the indirect addressing of entity index or variable index in GPSS.

[8] Scope of Entity and Variable

Variables and entities have their own scope of references. Their scope consists of the ACTIVITY in which they are declared and all encompassed ACTIVITY's that do not contain another declaration of the same identifiers. Any variable and any entity (including comact, but excluding ordinary actentity) declared in an ACTIVITY may be accessed in all ACTIVITY's encomppassed by it.

Once variables and entities declared in the definition of an ACTIVITY, they may be used in any ACTIVITY included in the subtree whose root is the ACTIVITY which their declaration. The variables and entities, which are used in an ACTIVITY of a comact and are not be local ones to the ACTIVITY, have to be declared in the root ACTIVITY of any one subtree which includes all the ACTIVITY's invoking the comact.

[9] Execution Specification Statement

Execution specification statements are used in order to construct the specification on generation and termination of transactions, the specification on initial attributes of transactions after its generation, and the specification on the condition of beginning and ending of a simulation run. They are GENERATE, TERMINATE, assignment, EXEC and STOP statements.

[10] Semantics of IMSS Modeling Language

Modeling statements are grouped into structure statements and non-structure statement. Non-structure statements are simple statements and compound statements. Compound statement is a sequnce of simple statements enclosed by BEGIN and END,

Simple statements are PUT, GET, USE, PREEMPT, IN, SET, RESET, HOLD, WAIT UNTIL, assignment and activity statements.

A transaction executing a PUT statement requests and reserves storage units whose number is evaluated from the given arithmetic expression. If the storage does not have enough available space for the request, the transaction must wait until the request is satisfied. A GET statement is used so as to remove and make available some previously occupied units.

A transaction executing a USE statement requests to use the specified facility. If it is being used by the same number of transactions as its capacity, the transaction must wait until one of them goes out of the facility. If not so, the transaction succeeds in using it, and stays in it for a time which is evaluated from the given expressions. A transaction executing a PREEMPT statement can preempt a facility which has already occupied by other transactions. This statement suspends the processing of the transaction most recently entering the facility.

A queue is used so as to collect statistics about the number of transactions which are waiting by a certain cause. An IN statement may be put just before a statement where there is an opportunity for transactions to be waiting such as a PUT, USE or WAIT UNTIL statement.

A transaction executing a HOLD statement stays in the present place for a time which is evaluated from the given expression.

A transaction execution a SET or RESET statement can always set or reset the specified semaphore regardless of its state.

A transaction executing a WAIT UNTIL statement waits until the boolean value evaluated from the given expression becomes true.

The value of the given arithmetic expression on the right hand of an assignment statement is assigned to a common or private variable on the left hand.

An activity statement is an ACT statement or an USE ACT statement. When a transaction encounters an ACT statement, it enters the ACTIVITY defining the action of the given actentity. When it reaches the exit point of this ACTIVITY, it returns to the previous ACTIVITY. When a transaction encounters a USE ACT statement, it enters the ACTIVITY defining the action of the given actentity after it has seized the given facility. After executing the ACTIVITY, it returns to the previous ACTIVITY and releases the facility. The difference between USE ACT and ACT statements exists in whether a transaction enters the given actentity conditionally or not.

Structure statements are IF, CASE and CONDITION statements for selection structures, REPEAT and DO statements for repetition structures, and SPLIT and MATCH statements for split structures.

If the boolean value of the given boolean expression in an IF statement is true, a transaction executes the then-clause, otherwise it executes the else-clause (if there is an else-clause) or the next statement (if not). If the value of the variable specified in a CASE statement is "i", a transaction executes the i-th statement. Pairs of boolean expression and compound statement are specified in a CONDITION statement. A transaction evaluates these boolean expressions one by one. If the value is true, its compound statement is executed.

A transaction repeats to execute the non-structure statement specified in a REPEAT statement until the specified boolean expression becomes true. While the value of the boolean expression specified in a DO statement is true, a transaction repeats to execute the specified compound statement.

In a SPLIT statement, one transaction is splitted into several transactions whose attributes are the same as those of the original transaction. The i-th transaction executes the i-th statement. After all transactions execute the respective statement, they are merged into one transaction, whose attributes are equal to the first split transaction. A MATCH statement serves to synchronize the progress of two or more splitted transactions. All transactions are allowed to proceed after the synchronization has been achieved.

Syntax of IMSS Language is shown in terms of Backus Naur form (BNF) in Appendix 1.

[11] IMSS Pictography

The second "language" in IMSS is named IMSS pictography. Pictographic symbols have the straightforward correspondence to IMSS modeling statements. A user can build his simulation model in IMSS pictography as well as in IMSS language. A pictographic form of a simulation model encourages a user to detect structual errors in his model.

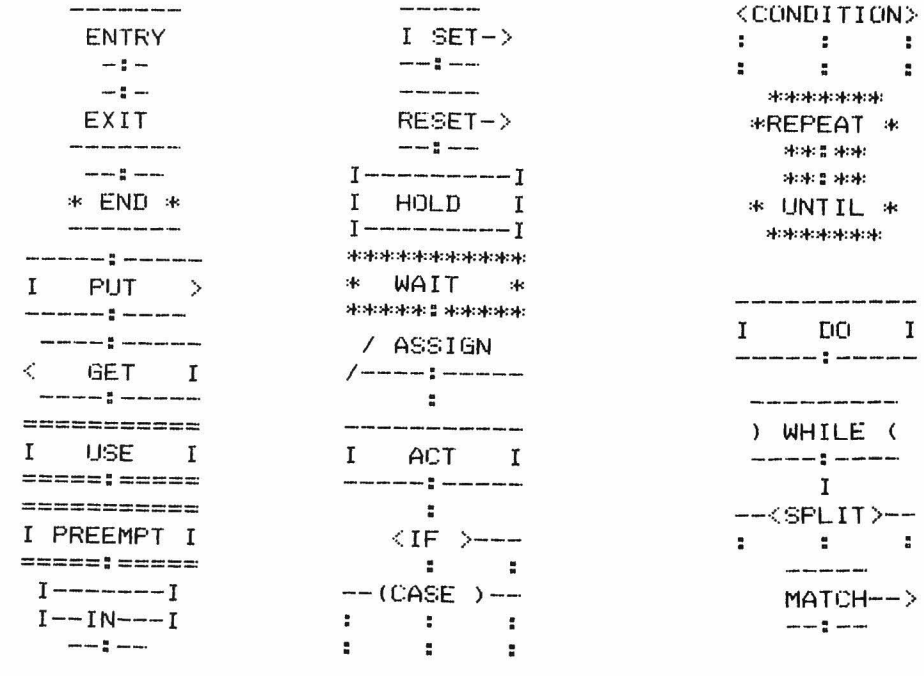


Fig.II.3.5 IMSS Pictography

II.3.2 Principles in Top-Down Modeling Process

Behavior of a system may be represented in one or more flows of transactions through parallel components of the model of the system. IMSS allows us to build a hierarchical model of a system with one ACTIVITY or several parallel ACTIVITY's each of which may be developed in a top-down fashion.

In the first stage of modeling a system, we decide whether the model of the system may be represented in one or more parallel components, and then provide transactions for the input of the component(s).

In the next or further stage, we construct the model of each component in the following top-down fashion.

(1) The function of the component of a system may be divided into several portions. These portions may correspond to elementary entities or actentities. The total number of portions should be limited so that we can write the entire textual model(or pictographic model) in about twenty lines (or about eight pictographic symbols) on a character display terminal.

(2) The flow control of the inputted transaction may be described as an ACTIVITY in a program text. An ACTIVITY has one entry and one exit only, and it is described using simple sequences of primitive, activity or structure statements.

These statements act on the transactions or the other types of entities. A structure statement has also one entry and one exit only. Note that any structure statement must not be used as a clause of another structure statement. Nesting of structure statements is not allowed. Subspecification -- specification or activity of the actentity corresponding to the activity statement -- is fully defined and its documentation is transmitted

to the next lower level of modeling stages.

(3) Substitute the word "actentity" for the word "component of the system" in steps from (1) to (2), and iterate the procedure in these steps until all of the actentities are finally represented by elementary entities.

Note the limitation of the number of statements or symbols. One-page model program allows a user to understand its structure at a glance.

Note the role of one entry and one exit of an ACTIVITY or a structure statements. Multiple entries and exits are potential source of confusion when reading and debugging a simulation model program.

Note the role of structure statements. Usual simulation languages allow us arbitrary sequence of primitive operators including a GOTO operator, and arbitrary sequence controls. Apparently such freedom in sequences and controls gives a great flexibility to modeling process, but the structure of the resulting model is apt to be so complicated that even model builder himself could easily read it, as well as anyone else. On the other hand, in our language the operations of flow control are restricted to only three types of structure statements (selection, repetition and split) execept the ordinary simple sequence. In fact, it is such the restriction that gives us a sort of conceptual discipline - how to construct a model step by step - in modeling process.

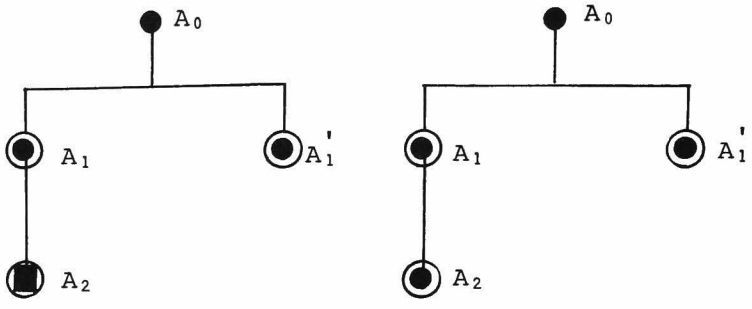
Variables and entities (including comacts) common to more than one parallel ACTIVITY's have to be declared in the model declaration part, i.e., the declaration part of the root of whole model tree including the parallel ACTIVITY's.

If a user wants to discontinue the above modeling process and to execute his model (which is not fully built) for its dynamic test, he may define the fuction of "actentity" as "simulation stub".

II.3.3 Simulation Stub

In IMSS, top-down process can be carried out not only in modeling but also in simulation execution. Model program in arbitrary stage in top-down process can be easily checked about syntax. Moreover, we can execute the model for its test, using "simulation stub", which simulates the presence of yet-to-be-implemented actentity. Simulation stub is given by the definition of ACTIVITY in simple or even sophisticated fashion at the desire of a user.

Fig.II.3.6 shows the simulation execution process by using a simulation stub. After we verify the functions of already-implemented activities A_0, A_1 and A_1 through simulation execution by using a simulation stub representing the functions of yet-to-be-implemented activity A_2 in simple fashion, we execute the complete simulation model with the fully defined activity A_2.




```
ACTIVITY A2;
$Simulation Stub$
PRIVATE P;
ENTRY;
 P=UNIFORM(a,b);
 HOLD P;
EXIT;
END A2
```

Simulation Execution using Simulation Stub

Verify Functions of Activities A_0, A_1 and A_1'

```
ACTIVITY A2;
STORAGE S;
FACILTY F;
PRIVATE P;
 P=UNIFORM(a,b);
 PUT xx TO S;
 USE F FOR P;
 GET xx FROM S;
EXIT;
END A2
```

Simulation Execution of whole model

Verify whole model and get statistics about model

Fig.II.3.6 Simulation Stub

Section II.4
Tool and Principle in Simulation Execution Mode

The most remarkable feature of IMSS is the capability of interactive monitoring of simulation execution of a hierachical simulation model which may includes none or more simulation stub. This capability is realized by various execution modes and reporting facility.

II.4.1 Interactive Execution

IMSS has five types of interactive execution modes as follows.

(1) Step-by-Step mode: Every time one model statement is executed, simulation run is suspented.

(2) Clock-by-Clock mode: Every time the simulation clock is updated, simulation run is suspended.

(3) Run mode: Simulation run is continued until the simulation clock time appointed by a user, and then a simulation run is suspended.

(4) Condition mode: Simulation run is continued until the stop condition, which is specified by an execution specification statement, is satisfied, and then simulation run is suspended.

(5) Roll-Back mode: Simulation run is returned to the previous simulation clock time appointed by a user, and then simulation run is suspended.

II.4.2 Reporting Facility

Simulation execution is performed in an arbitrary sequence of the above execution modes. When it is suspended, a user may obtain the state of a simulation model using reporting facility.

The reporting facility provides the capability to allow a user to inquire any statistics about transactions or entities according to the user's choice. Two types of reporting forms are provided, which are numerical form and line graph form.

The numerical form provides a decimal representation of statistics. The line graph form provides a representation of line graph with time axis on statistics. Various types of statistics are provided: "current contents", "average contents", "total entries", "average utilization rate", and "average use time per transaction" for a facility or a storage; all these types except for "average utilization", and in addition "zero entries ", which is the number of transactions passed without delay, for a queue; "generation/termination count", "mean duration time from generation to termination" for a transaction.

If a user wants to obtain one of statistics in the numerical form, he may specify a triple (a type of entity, an entity identifier, a type of statistics). A type of entity is facility, storage or queue. In order to make it easy to obtain several statistics, he may specify a simple (ALL), a double (a type of

entity, ALL), or a triple (a type of entity, an entity identifier, all). All statistics on all entities, all statistics on all entities whose type are identical and all statistics on a entity are outputted in the above three cases, respectively.

If a user wants to obtain one of statistics in the line graph form, he may do as well as the above. He may not specify the scaling of two axes of time axis and statistics axis. IMSS automatically determines their scaling according to the current simulation clock time and the maximum of statistics, respectively.

II.4.3 Principles in Interactive Execution Process

Principles in interactive monitoring are as follows. If a user wants to test his model, simulation run is performed in Step-by-Step mode, or in Clock-by-Clock mode. In the former mode, current values of entities are obtained before/after one model statement is executed. Therefore, a user can check his model microschopically at model statement level. In the latter mode, any statistics of entities are obtained before/after the simulation clock is updated. Thus, a user can check the behavior of his model every time any events occur. If the correctness of a model is guaranteed in the above way, the statistical property of the model may be obtained. The combination of numerical form of report and Run mode (or Roll-Back mode) allows a user to compare statistics at the beginning point of any interval of times with those at the ending point of the interval. The combination of line graph form of report and Run mode allows him to know whether a model is stable or not. If a user wants to obtain final statistics, simulation run may be performed in Condition mode.

Section II.5.
Command System

IMSS Command System controls the progress of simulation activity from modeling mode to simulation execution mode according to user's specification.(See Fig II.5.1.) In IMSS Command System, each mode in simulation activity is called IMSS mode. Five IMSS modes are provided, which are modeling mode, model translation mode, execution text construction mode, execution text translation mode, simulation execution mode.

In modeling mode, a simulation model is built using modeling subcommands, which specify editting functions such as appending, deletion, listing, and modeling statements. In execution text construction mode, an execution text is constructed using execution specification statement. In model translation mode and execution text translation mode, a simulation model and a execution text are translated to their internal form, respectively. If there are some syntax error in them, error message are outputted and the control must be returned to modeling mode and execution text construction mode, respectively. In simulation execution mode, simulation execution proceeds using subcomoands which specify interactive execution modes. When any mode of execution is completed, a user may inquire statistics on entities using subcommands which specify the form of reports. If simulation execution is found to errorneously proceed or the user wants to perform simulation execution with the minor modification to a model or an execution text, the control may be returned to modeling mode or execution text construction mode.

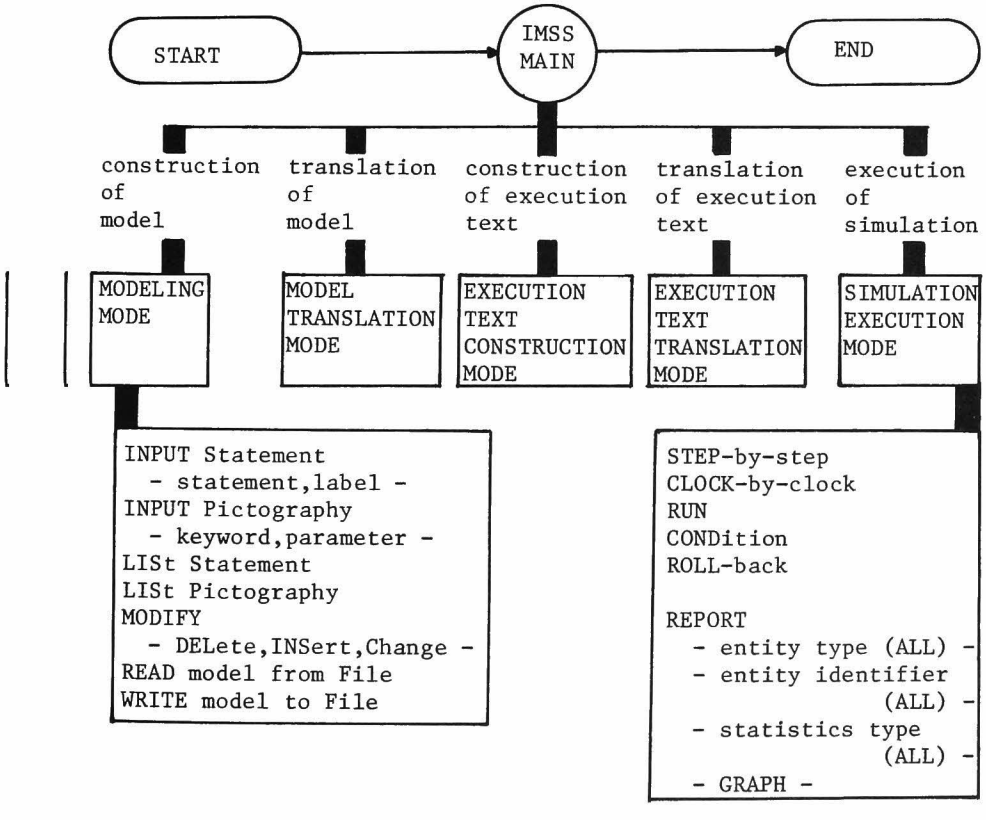


Fig.II.5.1 IMSS Modes and Commands

Section II.6
Implementation of IMSS

II.6.1 System Description

IMSS is implemented on the TSS of the large-scale computers FACOM M-190 and M-200 (OS IV/F4) in data processing Center of Kyoto University. A user may use IMSS with a character display/ teletypewriter terminal. IMSS is implemented in FORTRAN IV, so it has good portability. IMSS is transferred to HITAC M-160 in Systems Development Laboratory of Hitachi, LTD., where IMSS is a subsystem of integrated performance evaluation system named Interactive tool for System Configuration Planning(ISCP).

II.6.2 Organization of Simulator

IMSS simulator is composed of "command interpreter", "editor", "translator", "executer", "transaction scheduler", "transaction generator/terminator", "interpreter", "statistician", and" "reporter". (See Fig.II.6.1.)

"Command interpreter" accepts IMSS commands, interprets them and invokes "editor", "translator" or "executer".

"Editor" assists a user in building a simulation model or an execution specification text using editting subcommands, modeling statements and execution specification statements.

"Translator" translates an IMSS model into its internal form and it generates "simulation entity table" and kernels of "transaction scheduling list". "Translator" also translates an execution specification text into its internal form and it generates "execution control table" and "transaction generation/termination list".

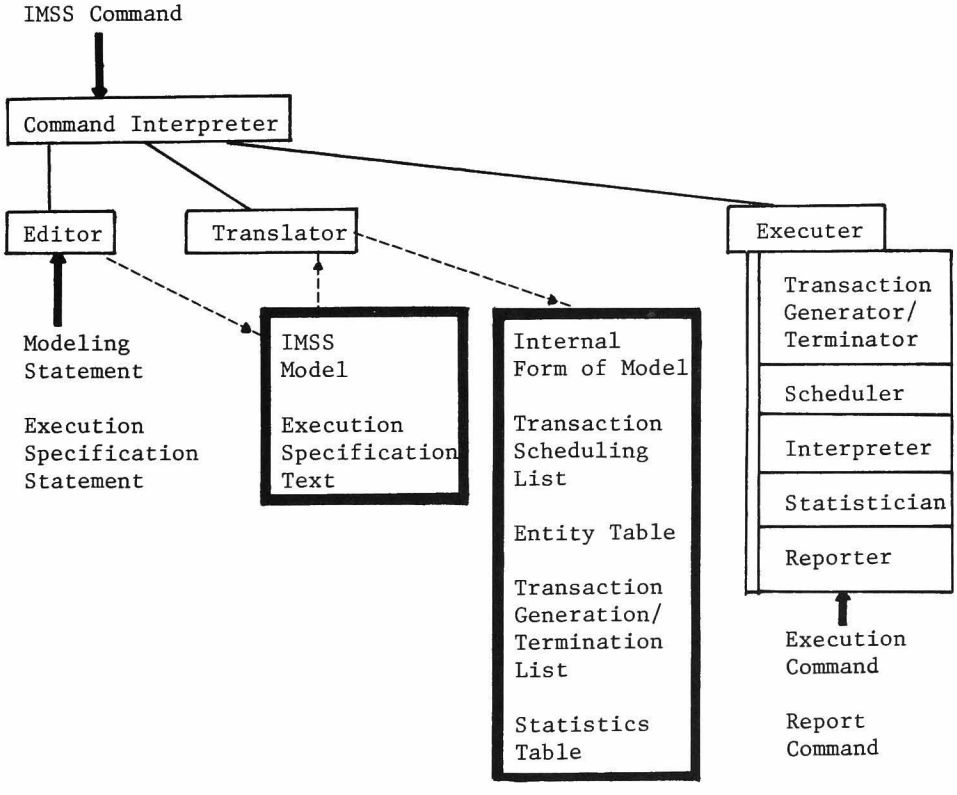


Fig.II.6.1 IMSS Simulator

"Executer" assists a user in monitoring the simulation execution process using execution subcommands and it manages the simulation execution process. "Transaction generator" creates transactions according to "transaction generation list". "Executer" registers the created transactions to the top of "transaction scheduling list" and selects one transaction which is moveable.

This transaction is passed to "interpreter" through "executer" and then "interpreter" moves the transaction over internal form of an IMSS model. During this process, several attributes of the associated entities in "simulation entity table" may be altered. Once transactions can not execute a certain operator in internal form of a simulation model - e.g., some transaction can not use a facility, because others have already occupied it -, "interpreter" relinquishes the movement of the transaction. "Executer" registers the transaction to the "transaction sceduling list". Again "transaction scheduler" selects another transaction and so forth. Transactions will be released to be moveable by some changes of the circumstances.

If any transaction arrives at the end of internal form of an IMSS model, it is passed to "transaction terminator" and goes out of existence.

II.6.3 Scheduling of Transaction

Transactions hold four states - active state, block state, ready state and hold state. (See Fig.II.6.2.) The active state is the state where a transaction is now running on the model. In the hold state, a transaction is consuming some period of time on a chain named a future chain.

If some transaction tries to access a certain entity (whose capacity is limited) when others have already occupied, or if it is awaiting the time when a certain condition is satisfied, it goes to the block state. On the other hand, in such a case that a transaction finishes to consume some period of time, that it is passed to ACTIVITY at the lower level, or that it comes back from the lower level ACTIVITY to the corresponding upper level actentity, it goes to the ready state.

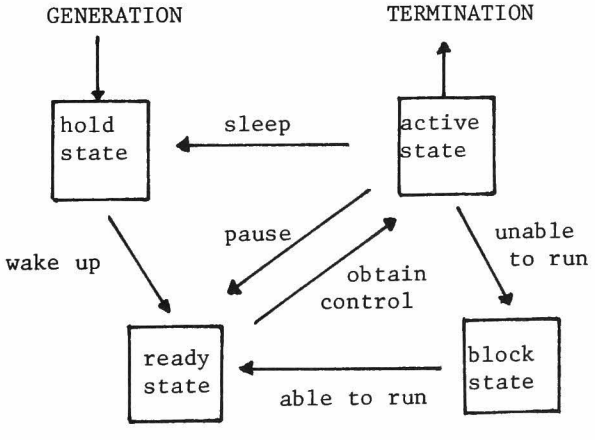


Fig.II.6.2 Transaction State and its Transition

The root of the block chain or the ready chain is created by "translator". This root is called a kernel element. The kernel element is created on the place where a transaction becomes to the ready state or the block state. If one or more kernel elements may access the same entity, these kernel elements share the same block chain attached to the entity. Kernel elements are connected with each other at each ACTIVITY. This chain of ACTIVITY is called a kernel.

The block chain is created at the following places.

a) PUT statement: This statement is used for the purpose of putting some quantity to the specified storage. As shown in Fig.II.6.3, a PUT statement is translated into its internal form by "translator". At the entry part of this internal form, a kernel element is created. That is, when a transaction goes to this place, it is immediately connected to the block chain of this kernel element and waits there until it can put some quantity to the storage.

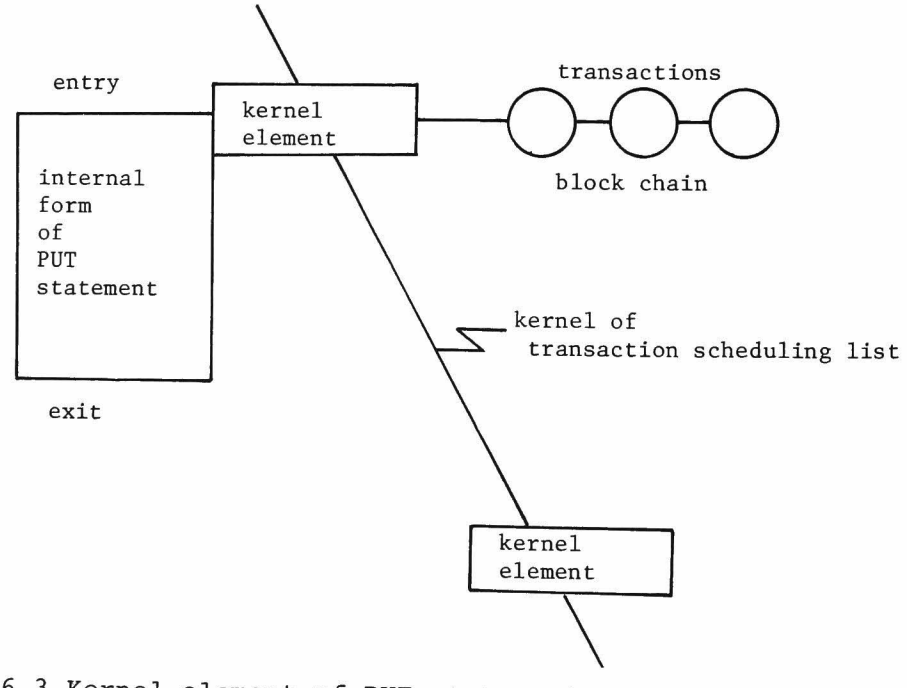


Fig.II.6.3 Kernel element of PUT statement

b) USE statement: A kernel element is creatd at the entry part of the internal form of a USE statement as well as in the case of a PUT statement.

c) WAIT statement: A kernel element is created at the internal form of a WAIT UNTIL statement. A transaction waits there until a condition specified in this statement is satisfied.

d) Exit of SPLIT statement: A SPLIT statement is composed of split specification part and several non-structure statements. Each of splitted transactions waits at the exit place of the SPLIT statement until all splitted transactions reach to the end of non-structure statements.

e) MATCH statement: All splitted transactions synchronize the progress with each other in a MATCH statement. When a transaction goes to the statement, it is immediately connected to the block chain.

The set of ready chains is created at the following places.

a) Entry of ACTIVITY: When a transaction tries to access some quantity, it is connected to the ready chain of a kernel element at the entry place of the corresponding ACTIVITY in Firt-In First-Out policy.

b) Actentity: Once a transaction finishes the execution in a certain ACTIVITY, it is connected to the ready chain of a kernel element at the corresponding actentity.

c) HOLD statement: Once a transaction finishes to consume some time, it is connected to the ready chain of a kernel element at the HOLD statement.

d) Entry of SPLIT statement: A SPLIT statement is used when more

than one transactions copied from one transaction move on each different nonstructure statement. At the place where transactions are splitted, a kernel element is created.

Each kernel element has its type and status words. The type indicates the class of the kernel element and its service policy. The first status word is the number of transactions connected to the kernel element. The second status word indicates whether the corresponding entity is available. If it becomes available, the status word is set "ON". Only kernel element corresponding to a facility or a storage has the second status word. On the other types of kernel elements, this word is considered to be always set "ON". "Transaction scheduler" inspects these status words and determines which transaction should be moved.

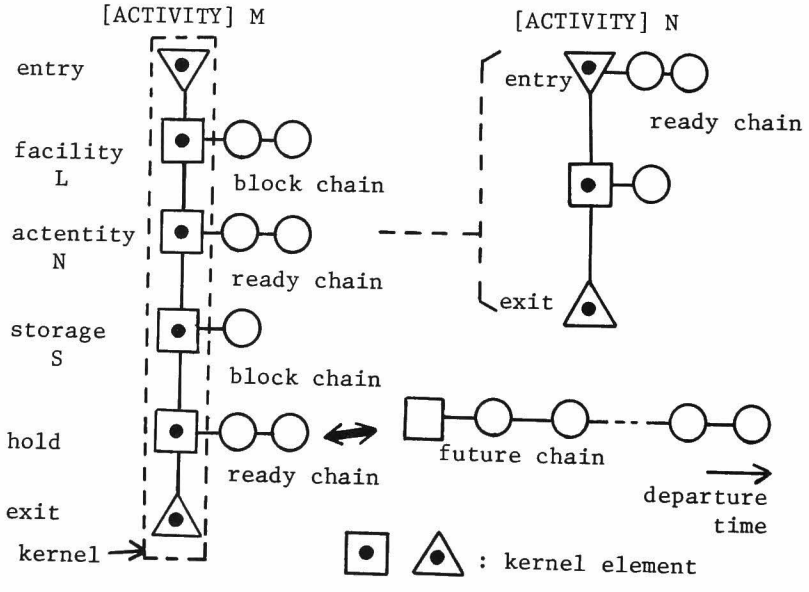


Fig.II.6.4 Transaction Scheduling List

Fig.II.6.4 shows the transaction scheduling list. Chains connected to kernel elements correponding to a facility "L" and a storage "S" are block chains. When a transaction comes to these places, it is connected to these chains according to the service policy of the corresponding entity.

Service policy means the order in which transactions act on the entity. Four types of service policy are First-In First-Out(FIFO), Last-In Last-Out(LIFO), Smallest-First-Out(SFO) and RANDom(RAND). If the service policy of a certain storage is FIFO or RAND, transactions line up in its block chain according to the order in which they come to the corresponding place. If it is LIFO, the order is reverse. If it is SFO, they line up in its block chain according to the order of access quantity.

Chains connected to kernel elements of a HOLD statement, entries of ACTIVITY's, and an actentity "N" are ready chains.

When a transaction enters into the actentity "N", it is connected to a ready chain of the lower ACTIVITY "N". If a transaction arrives at the exit of ACTIVITY "N", it is connected to the ready chain of the kernel element of the corresponding upper level actentity "N".

If a transaction gets to a HOLD statement, it is connected to the future chain. On the other hand, the transaction in the future chain whose departure time is equal to current clock time is connected to the ready chain of the kernel element of the HOLD statement.

Future chain is the chain where transactions consuming some periods of time are connected. It is a time-ordered chain of transactions according to their departure time. The departure time is the time when each transaction is waked up. The future chain is constructed in the following way.

At the beginning of simulation execution, some transactions are created from several generation lists. Attributes of this transaction are its generic number and its departure time. Here, departure time is the time when the created transactions start to run. These transactions are connected to the future chain as the hold state..

A transaction, whose departure time is equal to the value of current clock time, is connected to the ready chain at the entry of an ACTIVITY to be executed first or at a HOLD statement. On the former case another transaction is created from the corresponding generation list, and it is registered to the future chain.

If a transaction executes a HOLD statement, its departure time is calculated by adding consuming time to current clock time. According to the departure time, the transaction is connected to the future chain as the hold state.

Scanning of the transaction scheduling list is advanced in the following way. (See Fig.II.6.5.)

After simulation clock is updated, "transaction scheduler" starts to scan only kernel elements of ready chains, from the front element to the rear element in a certain kernel, and from the upper level kernel to the lower level kernel in the model. "Transaction scheduler" inspects the first status word of the kernel element. If the content of this word is not "0", "transaction scheduler" takes out transactions from the corresponding ready chain one by one, and passes each of them to "interpreter". This process is called top-down scanning.

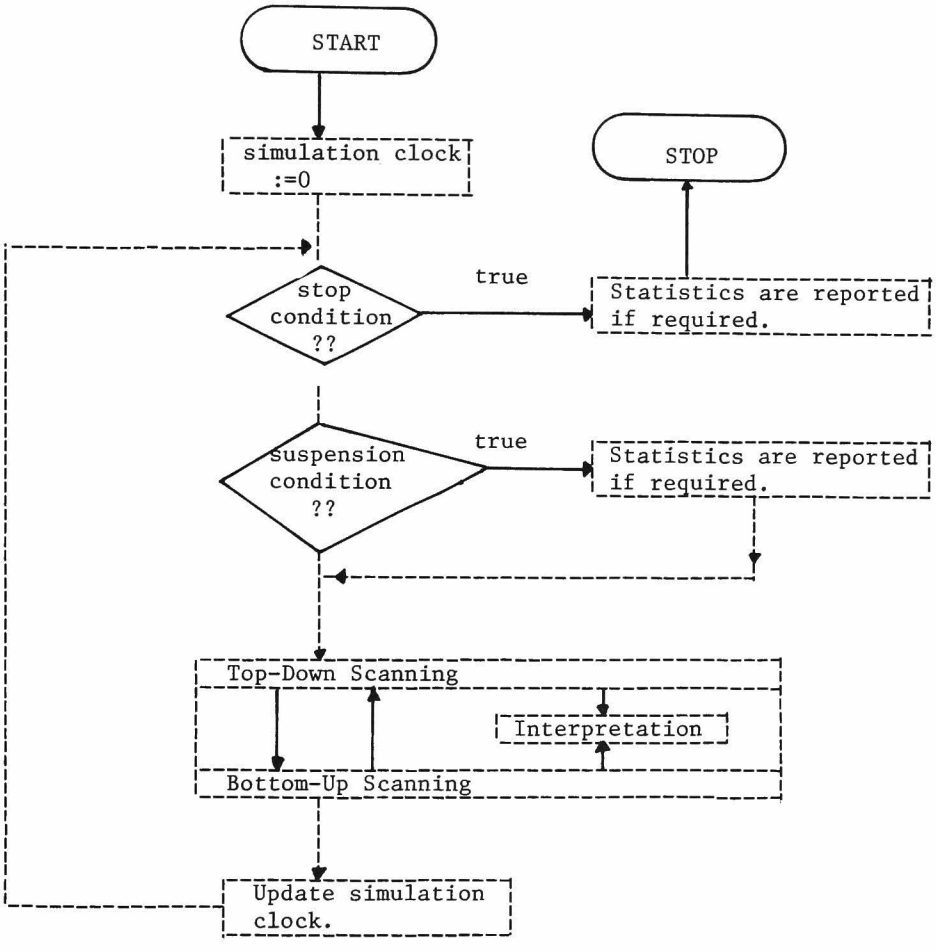


Fig.II.6.5 Simulation Execution Process

After top-down scanning, bottom-up scanning starts. "Transaction scheduler" scans all kernel elements, from the rear element to the front element in a certain kernel, and from the lower level kernel to the upper level kernel in the model. "Scheduler" inspects the two status word. If the content of the first word is not "0", and if the content of the second word is "ON", "scheduler" takes out one transaction from the corresponding ready chain or block chain, and passes it to "interpreter". Then, after interpretation, control is returned to "scheduler". "Scheduler" inspects the same kernel element. If the first status word is "0", or if the second status word is "OFF", the next kernel element must be scanned. Even if each of these words is not "0" or "OFF", when a transaction can not act on the corresponding entity whose service policy is LIFO, FIFO or SFO, the next kernel element must be scanned. In a kernel, this scanning is repeated until transactions to be moved do not exist. Top-down scanning and bottom-up scanning are repeated until moveable transactions do not exist in a model at current clock time.

The stop condition in Fig.II.6.5 means the condition on the termination of the simulation execution. It is specified by STOP statements in an execution specification text. The suspension condition means the condition by which the simulation execution is suspended in order to obtain statistics about a simulation model. It is always set "true" if the simulation execution procedes in Clock-by-Clock mode or it is set "true" when a simulation clock time becomes to the time specified by Run mode or Roll-Back mode.

II.6.4 Translation and Interpretation of Simulation Model

IMSS model is translated into an internal form text. Each IMSS statement is decomposed into a series of internal primitives. There are five types of internal primitives as follows. (See Table II.6.1.)

		mnemonic	parameter part
0	external number	EN	IMSS statement number in IMSS model
1	internal operator	IO	internal operator number
2	branch	B	relative location in internal form of text
3	branch on true	BT	relative location in internal form of text
4	get entity value	GETV	entity number
5	get entity location	GETL	entity number
6	get constant	GETC	constant
7	block	BL	kernel element number
8	no operation	NOP	------
9	get value of entity appointed in stack	GETSV	------
10	get entity capacity	GETCP	------
11	block to kernel element appointed in stack	BLI	------

Table II.6.1 Internal Primitives

(1) If a transaction comes to a certain internal operator(IO), the corresponding process is performed with the use of values on the stack. (See (3).) For example, when a transaction tries to access some storage, the corresponding process is performed with the use of storage location and access quantity on the stack. The type of an internal operator is designated by its parameter part. (See Table II.6.2.)

0	PLUS	+	arithmetic operator
1	SUBTRACT	-	
2	MULTIPLY	*	
3	DIVIDE	/	
4	EQUAL TO	=	boolean operator
5	NOT EQUAL TO	/=	
6	GREATER THAN OR EQUAL TO	>=	
7	LESS THAN OR EQUAL TO	<=	
8	GREATER THAN	>	
9	LESS THAN	<	
10	PUT	USE -- FOR -- PREEMPT -- FOR --	corresponding to IMSS statement
11	GET		
12	SEIZE		
13	RELEASE		
14	PREEMPT		
15	RETURN		
16	IN		
17	HOLD		
18	WAIT UNTIL		
19	SET		
20	RESET		
21	ASSIGN		
22	SPLIT		
23	MATCH		

Table II.6.2 Types of Internal Operators

(2) Two primitives change the order of execution in the internal form text. They are branch(B) and branch-on-true(BT) operators. The former uncoditionally changes the execution pointer of a transaction. The latter conditionally changes the execution pointer if the top value on the stack is true.

(3) Five primitives evaluate their own parameters or stack values, and put an evaluated value on the stack. Get-entity-value(GETV) primitive evaluates the current value of the entity indicated in the parameter part. Get-entity-location(GETL) primitive is used so as to know the location of the entity indicated in the parameter part. Get-constant (GETC) primitive puts the content, i.e., a constant, of the parameter part on the stack. Get-value-of-entity-appointed-in-stack (GETVS) primitive evaluates the current value of the entity indicated by the top value on the stack. Get-entity-capacity(GETCP) primitive is used so as to know the capacity of the entity located in the parameter part.

(4) Block(BL) primitive connects a transaction to a chain of a kernel element (located in the parameter part) and the transaction goes to the block state or ready state. Block-indirect(BLI) primitive connects a transaction, which has finished to execute some activity corresponding to a comact, to the ready chain at the comact regarded as an actentity with a kernel element number, where the transaction comes back. When a transaction tries to access a certain comact, its kernel element number is stored to a system-preserved private variable of the transaction. When a transaction executes BLI, the kernel element number has been already put on the stack. By using this stack value, the transaction can safely return to the calling ACTIVITY.

(5) External-number(EN) primitive is used for the purpose that the place of currently executing IMSS statement is displayed.

"Translator" is partitioned into parsing part and code generation part. The former determines the types of inputted statements. The latter generates internal forms according to parsing results.

Most of IMSS statements hold keywords such as PUT, USE, IF, CASE, etc. on their head. But only assignment statement does not hold a keyword. If we pay attention on this fact, translation process is in the following way.

First, the keyword is extracted from an inputted statement, and according to the syntax rule of the statement, the structure of the remainder of the statement is estimated. Then, through this estimation, the remainder will be parsed. If the estimation and the actual parsing result is matched, the parsing process is completed.

A primitive statement may be parsed by itself. On the other hand, a structure statement must be parsed through a series of statements. A structure statement is composed of its specification part - e.g., the specification part of an IF statement is IF clause and keywords such as THEN and ELSE - and compound statement part. Parsing of a structure statement is performed through the process that a group of simple statements is parsed after the specification part is parsed.

IMSS "interpreter" uses the slightly modified stack of Wheastone Compiler in Algol-60. The internal form not only of an expression but also of an IMSS simple statement is represented with the reverse Polish notation. The execution of an internal operator corresponding to an IMSS simple statement may be performed using some values on the stack. As a result, "interpreter" can be designed more simply.

Hereafter, this stack mechanism is explained with some examples.

Example-1) HOLD N;

A transaction is connected to the future chain by this statement. The internal form of this statement is shown as follows.

```
(a) GETV    N
(b) GETL    kernel element number
(c) IO      HOLD
```

The interpretation of (a) put a current value of a variable "N" on the stack. By (b), the kernel element number is put on the stack. A transaction will be connected to the ready chain of this kernel element after a transaction will consume a period of time appointed by "N". By (c), the execution is performed with two values of the stack. The transaction is registered on the future chain with a calculated departure time and a kernel element number.

Example-2) IN Q;

The internal form of this statement is shown as follows.

```
(a) GETL    Q
```

The interpretation of (a) puts the location of "Q" on the stack. This location is used by a succesive BL operator.

Example-3) P2=P1*3+6;

The internal form is shown as follows.

```
(a) GETL    P2
(b) GETV    P1
(c) GETC    3
(d) IO      MULTIPLY
(e) GETC    6
(f) IO      PLUS
(g) IO      ASSIGN
```

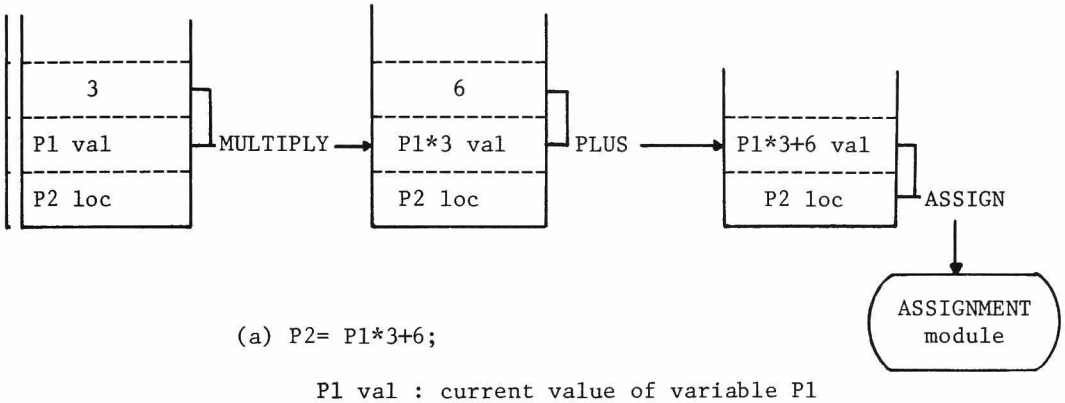
The interpretation process is shown in Fig.II.6.6(a). The interpretation of (a), (b) and (c) puts the location of a variable "P2", the current value of a variable "P1" and a constant "3" on the stack. After multiplication of two values of the stack, its result is put on the stack. Then, a constant "6" is put on the stack. Addition is performed and its result is put on the stack. Finally, assignment is performed.

Example-4) GET N from S(P1);

At this statement, a transaction gets some quantity(appointed by the current value of a variable "N") from a storage "S(P1)". Here, "S(P1)" means the p1-th element of a storage array "S". The internal form is shown as follows.

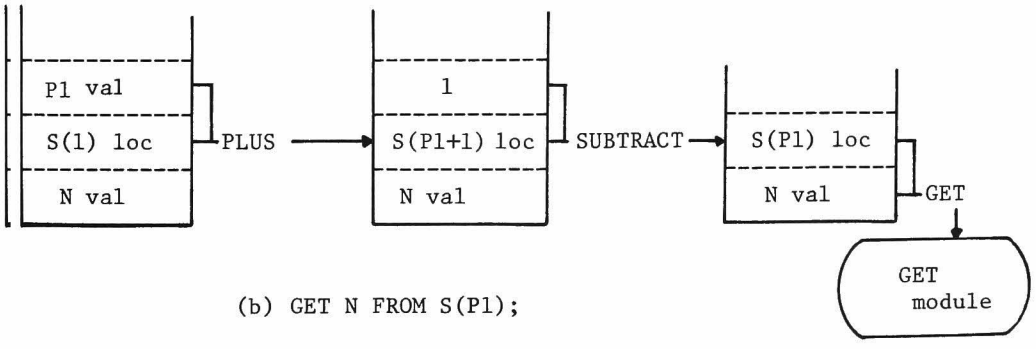
```
(a) GETV    N
(b) GETL    S
(c) GETV    P1
(d) IO      PLUS
(e) GETC    1
(f) IO      SUBTRACT
(g) IO      GET
```

In (a), the current value of a variable "N" is obtained. In (b), the location of the first element of "S" is put on the stack. After interpretation from (b) to (f), the location of P1-th element of S is put on the stack. The execution is performed with two values. This process is shown in Fig.II. 6.6.(b).



(a) P2= P1*3+6;

P1 val : current value of variable P1
P2 loc : location of variable P2



(b) GET N FROM S(P1);

Fig.II.6.6 Stack Mechanism in Interpretation

II.6.5 Collection of Statistics

On three entities such as storage, facility and queue, statistics are calculated. At each entity, IMSS has "number-of-entries", "accumulated-content", "last-number" and "current-number". "Last/current-number" means the number of transactions at the last/current simulation clock time. Last simulation clock time is the simulation clock time just before updated. As for storage, not "number" but "quantity" is provided. The statistics are computed in the following ways.

(1) Facilty:

"Number-of-entries" is the total number of transactions which have already entered into the facility.

"Accumulated-content"
= Σ (current-clock-time - last-clock-time) * last-number

"Average-content"
= accumulated-content / current-clock-time

"Average-utilization"= average-content / capacity

"Average-time-per-transaction"
= accumulated-contents / number of entries

(2) Storage:

"Accumulated-content"
= Σ (current-clock-time - last-clock-time) * last-quantity

The other statistics are calculated by the same method as those of facility

(3) Queue:

Statistics except for "average-utilization" are calculated by the same method as those of facility. Additionally, "zero-entries" is provided for the purpose to count the number of transactions passed through the queue without delay.

Fig.II.6.7 shows the method to collect queue statistics with stack mechanism. Some entity may be used at various places in the model. In that case, queue statistics are required at each place. For this purpose, a transaction puts the identifier of a certain queue on the stack, and then executes a BL operator. The transaction number and the identifier are registered to the block chain corresponding to the entity. IMSS inspects this block chain, calculates blocking time and registers it to the corresponding queue table.

On transaction entities, statistics are calculated at each group of transactions. A group of transactions is a collection of transactions which are generated by one GENERATE statement. A GENERATE statement determines the number of transactions to be generated, the simulation clock time at which the first transaction is generated, and the distribution of intergeneration time of transactions. Statistics on transaction entities are computed in the following ways.

"Number-of-generated-transactions" is the total number of transactions which have already been generated by one GENERATE statement.

"Number-of-terminated-transaction" is the total number of transactions which have already terminated.

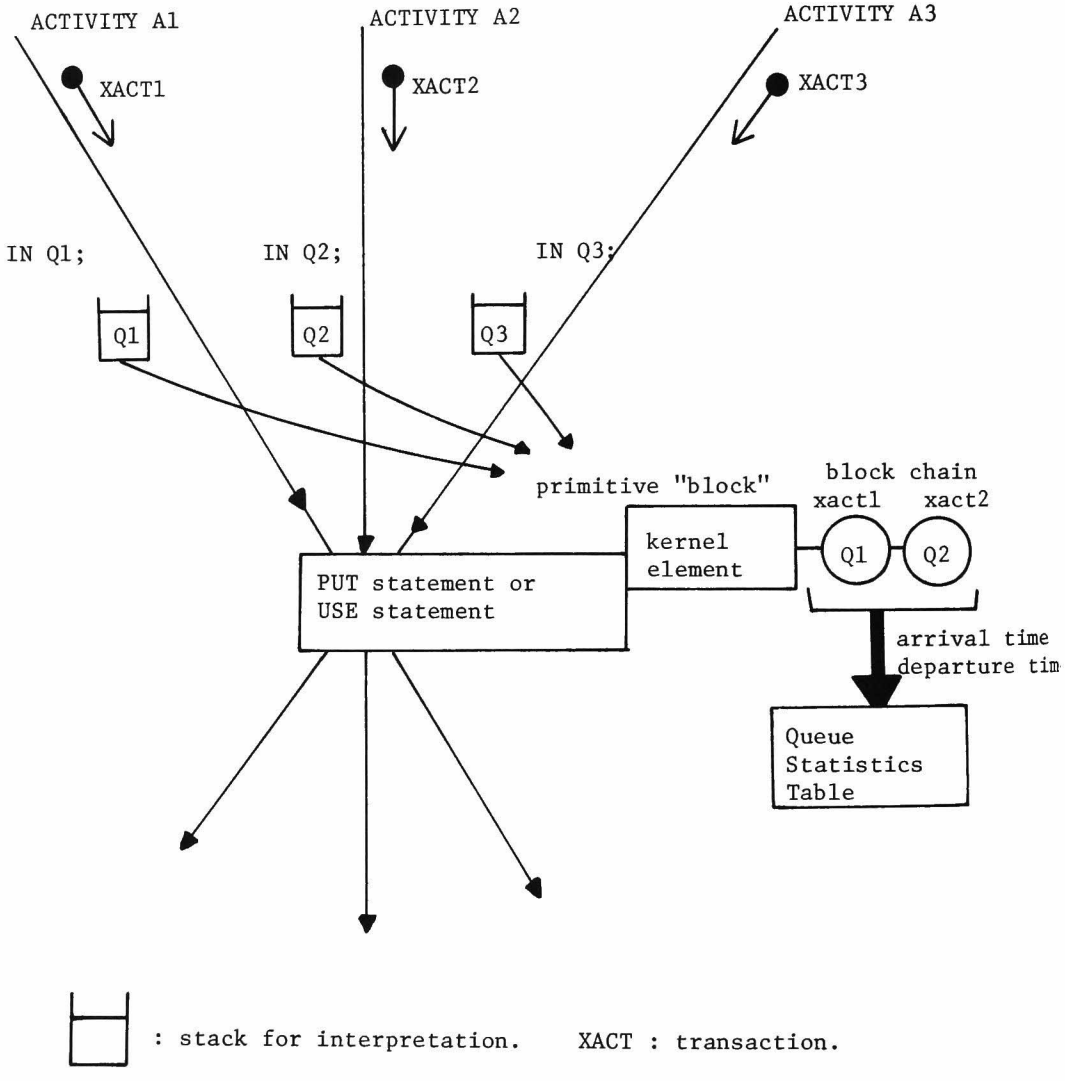


Fig.II.6.7 Collection of Queue Statistics with Stack Mechanism

The difference between the above two numbers is the number of transactions which exist in a simulation model when a simulation run is suspended.

"Average-duration-time" specifies the mean of intervals of times from the generation of transactions to their termination.

Section II.7

Examples

As the first example, we present a simulation model of a simple batch system where at most seven jobs use a CPU and a file repeatedly. Fig.II.7.1 shows the modeling process where a model is built in terms of IMSS modeling statements, IMSS pictography, modification commands, etc.. Fig.II.7.2 shows the whole structure of the model in terms of IMSS pictography. Fig.II.7.3 shows the monitoring process of the simulation model in terms of monitoring commands, numerical and line-graph form of statistics about simulation entities, etc..

```
******COMMAND MODE*********************COMMAND MODE******
SPECIFY,,,1-S,,2-P,,3-LIST,,4-MODIFY,,5-STOP MODELLING.
          6-READ F,7-WRITE F
01320 ?1
------SUBMODE INPUT:S -------------SUBMODE INPUT:S -----
06110 ? ENTRY;;
ENTER LABEL,IF NOT C/R
11070 ?
          1    ENTRY;


06110 ? USE BUFFER FOR JOB;
ENTER LABEL,IF NOT C/R
11070 ?
          2    USE BUFFER FOR JOB;


06110 ? EXIT;
ENTER LABEL,IF NOT C/R
11070 ?
          3    EXIT;
06110 ?
******COMMAND MODE*********************COMMAND MODE******
SPECIFY,,,1-S,,2-P,,3-LIST,,4-MODIFY,,5-STOP MODELLING.
          6-READ F,7-WRITE F
01320 ?2
------SUBMODE INPUT:P -------------SUBMODE INPUT:P -----
01380 ?ENTRY;
 4 ENTRY;
                                                    -------
                                                     ENTRY
                                                      -:-
```

Fig.II.7.1 Modeling Process of a Simple Batch System

(to be continued)

```
04660 ?
01380 ?USE CP
******COMMAND MODE*******************COMMAND MODE******
SPECIFY,,,1-S,,2-P,,3-LIST,,4-MODIFY,,5-STOP MODELLING.
          6-READ F,7-WRITE F
01320 ?2
------SUBMODE INPUT:P -------------SUBMODE INPUT:P -----
01380 ?USE
FACI.ID.-2=??
04520 ?CPU
ACTE.ID.-2  OR  ARITH.EXP. =??
04560 ?EXPO(50)
 9 USE CPU FOR EXPO(50);                          ==========
                                                  I   USE   I
                                                  =====:=====
ENTER LABEL,IF NOT C/R
04660 ?


01380 ?IN
NAME=??,IF NOT C/R
12890 ?Q2
 7 IN Q2;                                          I-------I
                                                   I--IN---I
                                                     --:--



 6 USE CPU FOR EXPO(100);                         ==========
                                                  I   USE   I
                                                  =====:=====
 7 USE CPU FOR EXPO(50);                          ==========
                                                  I   USE   I
                                                  =====:=====
 8 USE CPU FOR EXPO(100);                         ==========
                                                  I   USE   I
                                                  =====:=====


******COMMAND MODE*******************COMMAND MODE******
SPECIFY,,,1-S,,2-P,,3-LIST,,4-MODIFY,,5-STOP MODELLING.
          6-READ F,7-WRITE F
01320 ?4
------SUBMODE MODIFY---------------SUBMODE MODIFY------
11300 ?DEL 7 8
```

Fig.II.7.1 Modeling Process of a Simple Batch System

```
    1    IMSS QN(QUEUEING);
    2    ACTIVITY QUEUEING;
    3    ACTENTITY JOB;
    4    FACILITY BUFFER(7);
    5    QUEUE Q1;
 6 ENTRY;

 7 IN Q1;

 8 USE BUFFER FOR JOB;

 9 EXIT;

10 END QUEUEING;

   11    ACTIVITY JOB;
   12    QUEUE Q2,Q3;
   13    FACILITY CPU(1),FILE(1);
   14    PRIVATE COUNT;
```

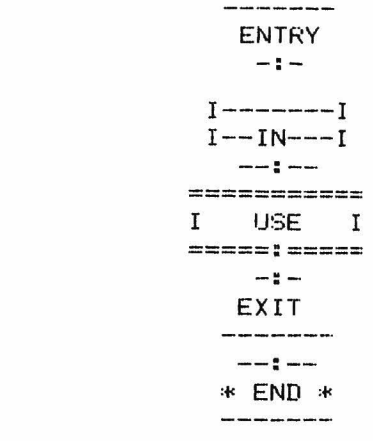


Fig.II.7.2 Whole Structure of Simulation Model of Fig.II.7.1

(to be continued)

```
15 ENTRY;                                  -------
                                             ENTRY
                                              -:-
16 COUNT=UNIFORM(4,3);                      / ASSIGN
                                           /----:-----
                                                :
17 IN Q2;                                   I-------I
                                            I--IN---I
                                              --:--
18 USE CPU FOR EXPO(100);                  ===========
                                           I   USE   I
                                           =====:=====
19 REPEAT BEGIN;                             *******
                                            *REPEAT *
                                              **:**
20 IN Q3;                                   I-------I
                                            I--IN---I
                                              --:--
21 USE FILE FOR EXPO(25);                  ===========
                                           I   USE   I
                                           =====:=====
22 IN Q2;                                   I-------I
                                            I--IN---I
                                              --:--
23 USE CPU FOR EXPO(100);                  ===========
                                           I   USE   I
                                           =====:=====
24 COUNT=COUNT-1;                           / ASSIGN
                                           /----:-----
                                                :
25 END;                                       --:--
                                             * END *
                                             -------
26 UNTIL L1:(COUNT=0);                         **:**
                                            * UNTIL *
                                             *******
27 EXIT;                                       -:-
                                              EXIT
                                             -------
28 END JOB;                                    --:--
                                             * END *
                                             -------
29 END IMSS;                                   --:--
                                             * END *
                                             -------
```

Fig.II.7.2 Whole Structure of Simulation Model of Fig.II.7.1

```
??EXECUTION MODE=(STEP/CLOCK/RUN/GRAPH/COND/TERM/ROLL)
10820 ?RUN 10000
**************************************
** MODE RUN 10000                    **
**************************************
******************************************
** CURRENT SIMULATION TIME :       11218 **
******************************************
??ENTITY=
 F-FACILITY,,S-STORAGE,,Q-QUEUE,,ALL-ALL,,C/R-EXIT
34200 ?F
??IDENTIFIER=    /    ALL-ALL,,   C/R-RETRY
34800 ?CPU
??STATISTICS=
 1-CURRENT CONTENTS,,2-ACCUMULATED CONTENTS,,3-AVERAGE CON
 5-AVERAGE UTILIZ.,,6-AVERAGE TIME/TRANS.,,7-MAX CONTENTS,
 9-ZERO ENTRIES,,10-PERCENT ZEROS,,11-$AVERAGE TIME/TRANS.
 ALL-ALL,,C/R-RETRY
40100 ?ALL
**     2 ** CPU          ** FACILITY **
      CURRENT CONTENTS=        1.00

      ACCUMULATED CONTENTS=    6180.00

      AVERAGE CONTENTS=        0.55

      TOTAL ENTRIES=       60.00

      AVERAGE UTILIZ.=         0.55

      AVERAGE TIME/TRANS.=     103.00

      MAXIMUM CONTENTS=        1.00

      CAPACITY=        1.00
??EXECUTION MODE=(STEP/CLOCK/RUN/GRAPH/COND/TERM/ROLL)
10820 ?CLOCK
**************************************
** MODE CLOCK                        **
**************************************
******************************************
** CURRENT SIMULATION TIME :       11308 **
******************************************
??ENTITY=
 F-FACILITY,,S-STORAGE,,Q-QUEUE,,ALL-ALL,,C/R-EXIT
34200 ?
??EXECUTION MODE=(STEP/CLOCK/RUN/GRAPH/COND/TERM/ROLL)
10820 ?ROLL 0
**************************************
** MODE ROLL 0                       **
**************************************
******************************************
** CURRENT SIMULATION TIME :             0 **
******************************************
```

Fig.II.7.3 Monitoring Process of Simulation Execution

(to be continued)

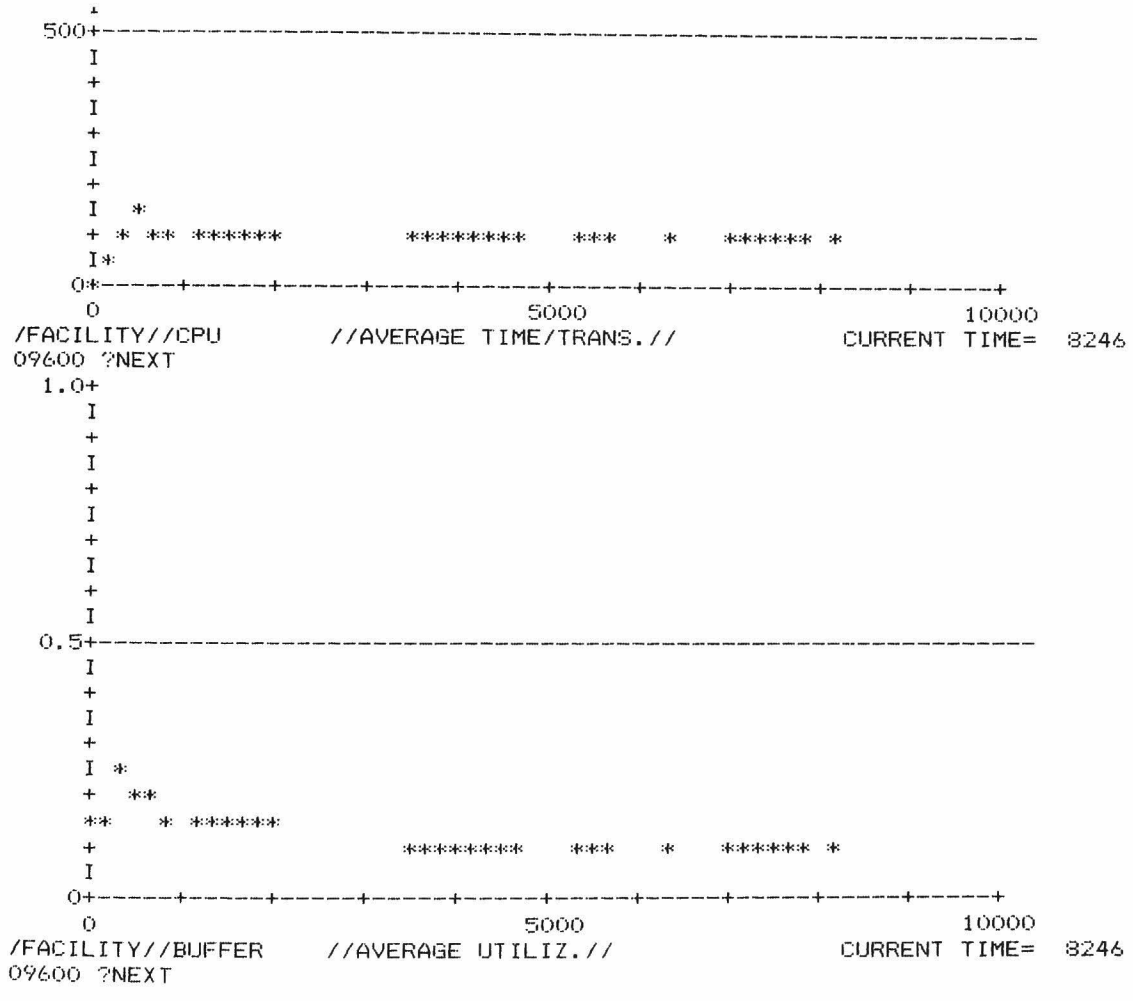


Fig.II.7.3 Monitoring Process of Simulation Execution

As the second example, we present a simulation model of an on-line sales order entry system. The purpose of the system is to allow salesmen to type order transactions and cancel transactions from their terminals, warehouse clerks to input arrival transactions and shipment transactions from their terminals, and accounting clerks to input charge transactions and payment transactions from their terminal. We are interested in various utilizations of resources such as a CPU and a file channel.

Fig.II.7.4(a)-(d) show the top-down modeling process. Fig.II.7.4(a) shows the main program(the root ACTIVITY "MAIN"), which simulates the task controlling the flow of each transaction ("CASE SORT OF 6;"). Fig.II.7.4.(b) shows the flow of arrival transactions. In this ACTIVITY("ARRIVAL"), arrival transactions enter the task updating an inventory-order file ("USE INVORD FOR C_INVORD;"). Fig.II.7.4(c) shows this task(ACTIVITY "C_INVORD") Fig.II.7.4(d) shows the flow of shipment transactions, which is implemented as a simulation stub. Fig.II.7.4(e)-(h) shows the monitoring process of the model including the simulation stub. Fig.II.7.4(e) shows Run mode of simulation run. Fig.II.7.4(f) shows statistics about CPU in the numerical form. Fig.II.7.4(g) shows statistics (average utilization) about file channel in the line graph form. Fig.II.7.4(h) shows statitics about transactions.

```
12   ENTRY;                                          --------
                                                       ENTRY
                                                        -:-
13     USE BRANCH FOR A_BRANCH;                      ===========
                                                     I   USE    I
                                                     =====:=====
14     CASE SORT OF 6;                               --(CASE )--
                                                     :    :     :
                                                     :    :     :
   15       M1: ACT ORDER;
   16       M2: ACT CANCEL;
   17       M3: ACT SHIPMENT;
   18       M4: ACT ARRIVAL;
   19       M5: ACT PAYMENT;
   20       M6: ACT CHARGE;
21     USE BRANCH FOR A_UNBRANCH;                    ===========
                                                     I   USE    I
                                                     =====:=====
22   EXIT;                                              -:-
                                                       EXIT
                                                       -------
```

(a) ACTIVITY "MAIN" of a Model of a Sales Order Entry System

```
   44   ACTIVITY ARRIVAL;
45   ENTRY;                                          -------
                                                       ENTRY
                                                        -:-
46     USE BRANCH FOR C_RELINK;                      ===========
                                                     I   USE    I
                                                     =====:=====
47     USE INVORD FOR C_INVORD;                      ===========
                                                     I   USE    I
                                                     =====:=====
48   EXIT;                                              -:-
                                                       EXIT
                                                       -------
                                                        --:--
49 END ARRIVAL;                                       * END *
                                                       -------
```

(b) ACTIVITY for Flow of Arrival Transactions

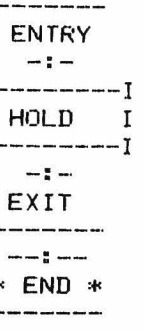
Fig.II.7.4 Modeling and Monitoring Process

(to be continued)

```
67   ACTIVITY C_INVORD;
68     PRIVATE COUNT;
69     ENTRY;
70       CASE SORT OF 4;
71       INVORD1: COUNT=4;
72       INVORD2: COUNT=4;
73       INVORD3: COUNT=5;
74       INVORD4: COUNT=2;
75       REPEAT BEGIN;
76          USE CPU FOR 3;
77         USE FILE1 FOR UNIFORM(40,35);
78         USE CPU FOR 1;
79         COUNT=COUNT-1;
80         END;
81       UNTIL INVORD5:(COUNT=0);
82     EXIT;
83   END C_INVORD;
```

(c) ACTIVITY for Flow of Transactions in a File Updating Task

```
  110   ACTIVITY SHIPMENT;
  111   $ SIMULATION STUB FOR SHIPMENT $
** ENTRY;

**    HOLD 150;

** EXIT;

** END SHIPMENT;
```



(d) Simulation Stub for Flow of Shipment Transactions

Fig.II.7.4 Modeling and Monitoring Process

(to be continued)

```
??EXECUTION MODE=(STEP/CLOCK/RUN/GRAPH/COND/TERM/ROLL)
10820 ?RUN 10000
*****************************************
** MODE RUN 10000                       **
*****************************************
*****************************************
** CURRENT SIMULATION TIME :       10002 **
*****************************************
```

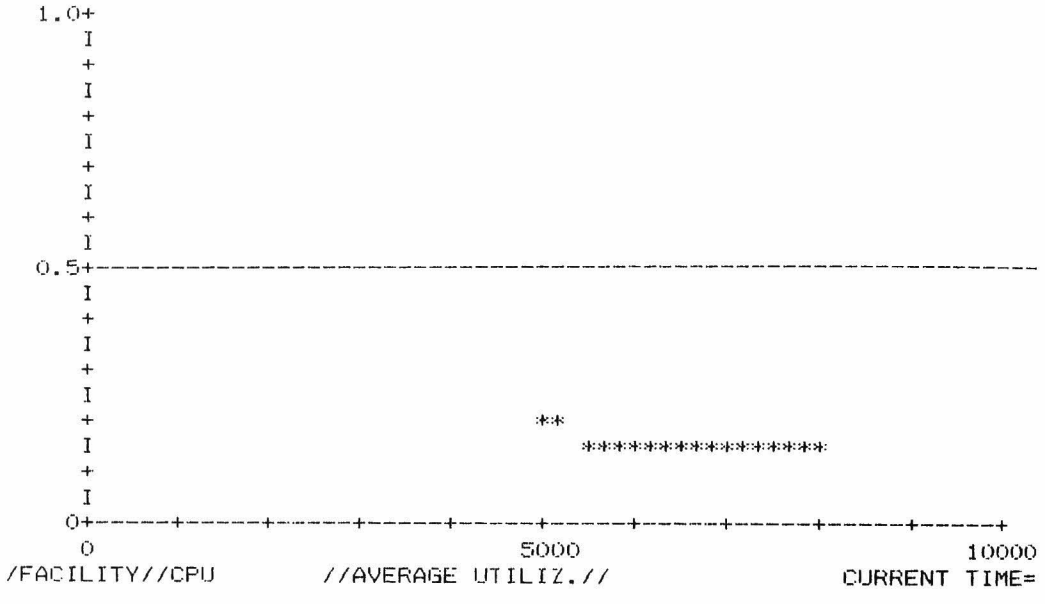
(e) Run Mode of Simulation Execution

```
??ENTITY=
 F-FACILITY,,S-STORAGE,,Q-QUEUE,,ALL-ALL,,C/R-EXIT
34200 ?F
??IDENTIFIER=    /    ALL-ALL,,   C/R-RETRY
34800 ?CPU
??STATISTICS=
 1-CURRENT CONTENTS,,2-ACCUMULATED CONTENTS,,3-AVERAGE CONTENTS,,4-TOTAL
 5-AVERAGE UTILIZ.,,6-AVERAGE TIME/TRANS.,,7-MAX CONTENTS,,8-CAPACITY
 9-ZERO ENTRIES,,10-PERCENT ZEROS,,11-$AVERAGE TIME/TRANS.
 ALL-ALL,,C/R-RETRY
40100 ?ALL
**     4 ** CPU          ** FACILITY **
      CURRENT CONTENTS=        0.0

      ACCUMULATED CONTENTS=    1797.00

      AVERAGE CONTENTS=        0.18

      TOTAL ENTRIES=     747.00

      AVERAGE UTILIZ.=        0.18

      AVERAGE TIME/TRANS.=        2.41

      MAXIMUM CONTENTS=        1.00

      CAPACITY=        1.00
```

(f) Statistics about a Facility in Numerical Form

Fig.II.7.4 Modeling and Monitoring Process

(to be continued)



(g) Statistics about a Facility in Line Graph Form

*** MEAN FLOW TIME ***

GENERATION LIST	GENERATION COUNT	TERMINATION COUNT	MEAN FLOW TIME	STANDARD DEVIATION	MAX FLOW TIME
1	54	49	607.408	458.999	1822
2	11	9	510.000	394.272	1220
3	54	48	1094.000	508.544	2275
4	6	4	417.250	205.423	658
5	40	39	340.282	193.721	983
6	11	9	357.444	208.980	843

(h) Statistics about Transactions

Fig.II.7.4 Modeling and Monitoring Process

Section II.8
Conclusion

We have developed the Interactive Modeling and Simulation System (IMSS) on the large-scale computers FACOM M-190 and M-200. IMSS introduces "online simulation process" and "top-down modeling and simulation execution process" into the simulation activity.

IMSS is a general purpose and transaction-oriented simulation system. IMSS has two types of modeling "languages" i.e., IMSS language and IMSS pictography. IMSS language is based on the Structured Programming technique which is well fit for top-down modeling and simulation execution. Top-down modeling is performed using "actentity", "common actentity" and "ACTIVITY". We can test or execute a simulation model using "simulation stub" at the arbitrary stage in top-down modeling. IMSS allows us to monitor or interact with the execution process of a simulation model using various execution modes and reporting facility. IMSS Command System accepts IMSS commands and swiches one mode to another in simulation activity. Simulation activity is performed effectively and easily due to these facilities of IMSS.

Top-down modeling and simulation execution process is well fit for the simulation of a software system which is being developed in top-down fashion. In order to obtain the assurance about the valid performance of the software system, the software designer can build a hierarchical simulation model at the same level of developing the object software system and he can execute the model by using "simulation stub".

Chapter III
System Description and Evaluation System: SDES

Section III.1
Introduction

When a software system is being developed,functional verification and performace evaluation must be performed continually and at all stages. Verification and evaluation are required both to rapidly and smoothly advance the system development and to ensure the system reliability.

When a designer designs a software system in detail, functional verification and performance evaluation are very important factors in correcting time-dependent errors and erroneous interrelationships between software modules before they reach the implementation stage. Cleary, we need a useful functional verfication and performance evaluation tool for software systems being designed in detail.

Usually, a software system consists of one or more concurrent processes [Hansen 1970 1972 1973 1974]. Each concurrent process is an autonomous entity which performs sequential processing and, if necessary, cooperates with other concurrent processes. Concurrent processes are sometimes called cooperating sequential processes [Dijkstra 1965 1968a]. Functional verification of a software system means verification for the logical behavior of each process and the logical interrelationships between processes. The logical behavior of each process includes the

processing steps in which the process behaves. The logical interrelationship between processes is referred to as synchronization, communication, congestion or competition. Performance evaluation for the software system includes evaluation of the time-dependent behavior of each process and the time-dependent interrelationships between processes under the actual environment. Time-dependent behavior is, for example, processing time which the process uses in its behavior. The time-dependent interrelationships between processes is, for example, waiting time which each process uses in synchronization, communication, congestion or competition.

In detailed designing, the software system is often written in certain programming language, but there exist few systematic methodologies for functional verification and performance evaluation of software systems whose details are being designed.

There are two types of functional verification, static and dynamic. The type of static verification most used is so-called design review. But, it is not easy to read a description of a software system in design review because the system consists usually of several concurrent processes. Verification in terms of an assertion is another method used. This method is based on the Mathematical Theory of Computation. An assertion is an invariant condition for the range of values of a variable or the relations between several variables. But, it is often difficult to make such an assertion apply to the description of a software system because variables may be shared by several processes concurrently. Verification in terms of a tracer or debugger is viewed as a method of dynamic verification. This method also presents difficulties because output from concurrent processes are often interleaved.

As for performance evaluation, the methods most used are the analytical and simulation methods. One of the most popular

analytical methods is a method using a queuing model based on the Queuing Theory. A queuing model is constructed by expressions representing the statistical distribution of interarrival times, service times, etc. In simulation, a simulation model is constructed by using a transaction-oriented simulation language such as GPSS or an event-oriented simulation language such as SIMSCRIPT. These two types of models for the performance evaluation are constructed in two phases. The first phase is the abstraction of functions of the object software system and the second phase is the addition of performance information. Languages and notations used to construct the models are different from those used to develop the object software system. Constructing the models and the object software system are performed separately takes a lot of time and effort. In addition, the separation mentioned above tends to generate errors and make specifying correspondence between the object software system and its model difficult. While an analytical or simulation method may be suitable for evaluating the performance of a software system at the design stage of overall structure of the software system, it is not suitable for the detailed design stage.

We have developed the System Description and Evaluation System(SDES) to verify the functions of a software system and evaluate its performance effectively at the time the system details are being designed [Itoh 1977b 1977c 1978a 1979a 1979b] [Tabata 1978]. In SDES, the description of the software system is integrated with the desciption used for its fuctional verification and performance evaluation. This single integrated description is then executed to verify and evaluate the software system with the aid of a computer. The detailed design, and verification and evaluation of a software system in SDES are performed in the following steps.

First, the software system is designed in PL/I. Then, the description of the behavior of the entities to be processed is

overlapped with the design description of the software system. Finally, the SDES verification/evaluation system is used to execute the design description and verify and evaluate the software system in terms of the description of the behavior of entities to be processed. We call the method by these steps the "Traversing Method".

A software sytem written in PL/I describes the processing steps of each concurrent process and the interrelationships between them. The design description is an executable program, and the level of detail it includes is arbitrary. The arbitrariness of the detail of the description means that SDES can be applied to the design of a software system from the beginning through the final stages.

The entity to be processed is, for example, input data, messages and transactions. The description of the behavior of these entities consists of the description of paths on which the entities move and the description of the interaction between the entities and variables/resources used in the software system.

Section III.2
Basic Concept of SDES: Traversing Method

The design description of a software system which is being constructed is named a "design description text", and may include several processes. A design description text is an executable text in which each control structure of processes and interprocess communication between them are fully specified, but in which all processing details of processes may not be fully specified. A design description text which contains few details of processes is called a "skeleton program", especially [Schechter 1978]. The text is written in a high-level programming language, such as PL/I with multi-task option, Concurrent Pascal [Hansen 1977a 1977b], Modula [Wirth 1977a 1977b], or Concurrent Lisp [Masaki 1978] [Tabata 1979] all of which can represent concurrent processes. In the first version of SDES, it is written in PL/I with multi-task option.

A software system being executed may have one or more processing flows. There are two opposite types of entities in a processing flow. The first is a "processing entity", and the second is an entity to be processed, named a "processed entity". The first processes the second according to the design description of the software system.

A processing entity is a processor, which may be a physical entity such as a central processing unit(CPU) or an input/output channel, or a logical entity such as a process. Synonyms for a process are a task and an activity. A processed entity is, for example, input data, a message or a transaction.

When constructing a design description text, processing steps of a processing entity and interrelationships between processes are described algorithmically in PL/I. One or more processing entities may exist, and each processing steps of the processing entities may be described separately.

Conceptually, each processing entity is considered to be autonomous, performing sequential processing on its own variables and resources and, if necessary, cooperating with other processing entities in terms of shared variables and resources. Such an entity is called a "process". Cooperation between processes involves transferring a processed entity from one process to another.

In the actual environment, processing steps of an processing entity may be overlapped in time or be interleaved with processing steps of other processing entities. These processing entities are called "concurrent processes".

In verifying the functions of a software system, we must verify the processing steps of each process and verify that the flow of each processed entity are realized so validly that the processed entity transfers between processes.

While processes performs their sequential processing according to their processing steps, they may compete to perform processing to variables or resources shared with each other. Therefore, mutually exclusive access to shared variables or resources must be guaranteed. While a process performs its processing to a processed entity, it accesses to shared variables and resources in competition with the other processes. It is effective to make mutually exclusive access to shared variables and resources is verified in terms of a processed entity.

Transaction-oriented simulation activity is used when evaluating

the performance of an object system as it is being designed. When constructing a simulation program, the behavior of processed entities in the system is described as the behavior of transactions in a simulation programming language such as GPSS.

The usual program is constructed as follows. A process STOREs (LOADs) a message to (from) a memory area; a process SENDs (RECEIVEs) a message to (from) the other process. The GPSS simulation program corresponding to the above is constructed as follows. A message ENTERs(LEAVEs) a storage area; a message TRANSFERs from one process to another process. (See Table III.2.1.)

Example 1 A process STOREs a message to a memory area. *A message ENTERs a memory area.*
Example 2 A process LOADs a message from a memory area. *A message LEAVEs a memory area.*
Example 3 A process SENDs a message to another process. A process RECEIVEs a message from another process. *A message TRANSFERs from one process to another process.*

Statements in usual programs are written in normal letters.
Statements in simulation programs are written in italic letters.

Table III.2.1 Differences between Usual programs and Simulation Programs

Since two opposite concepts, i.e., a processing entity and a processed entity, are separately applied to the programming activity and the simulation activity, errors may occur in transforming from one to another, or proper correspondence between them may be lost. The upper half of Fig.III.2.1 shows a tradional method used for evaluation in terms of the simulation activity.

We intend to integrate two opposite concepts into one description in both functional verification and performance evaluation for a software system. We overlap the description of the behavior of processed entities to the design description text, i.e., the executable description of the processing steps of processing entities. The SDES verification/evaluation system verifies the function of the design description text and evaluates its performance with the aid of the description of the behavior of processed entities.

The processing flow of individual entities is sequential (and partially in parallel). Many processed entities may be processed in parallel. Overlapping makes the individual behavior of each processed entity visual on the original design description text.

The Traversing Method allows us to identify this multiple behavior of processed entities in an original design description text, to overlap the description of the behavior of processed entities to the text, to verify the the function of the original design, and to evaluate its performance. A traverser is a unit which represents a processed entity and is used to trace its behavior. The individual behavior of each processed entity is written as the traverser description. A collection of traverser descriptions is used for verifying the function and evaluating the performance of the design description text for concurrent processes.

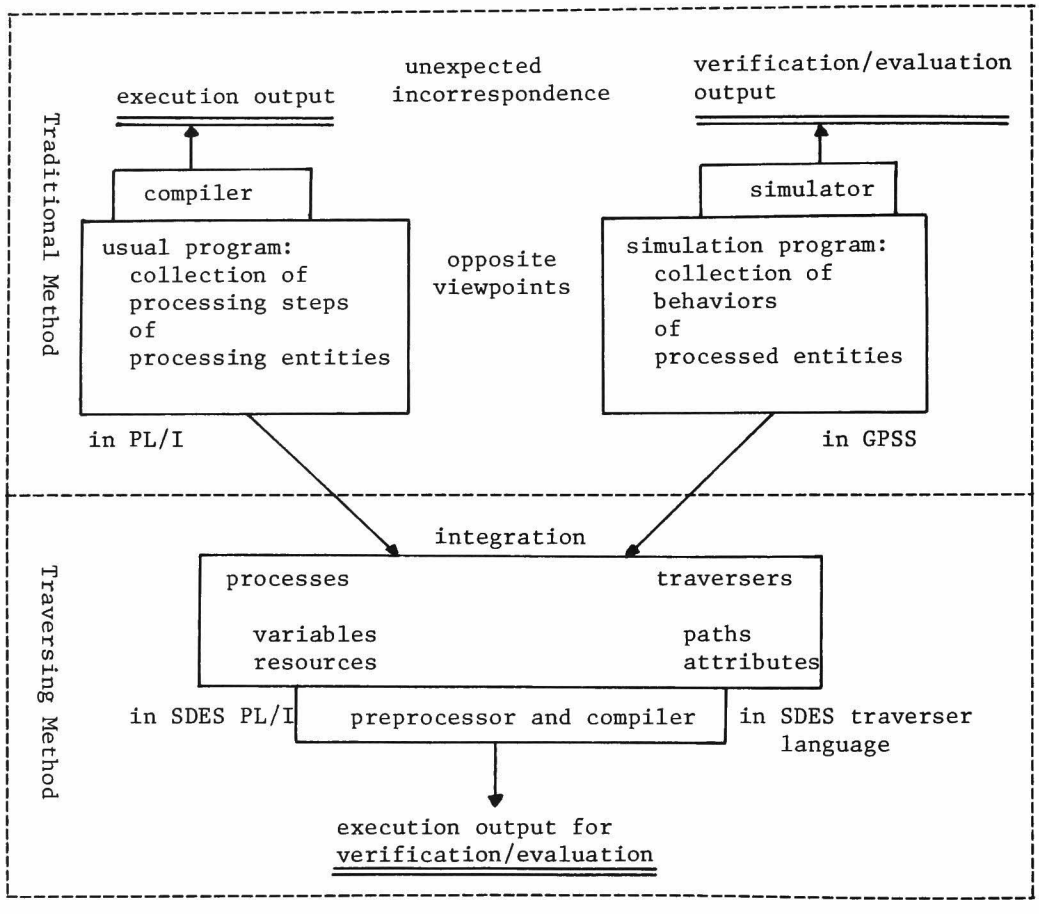


Fig.III.2.1 Traversing Method versus Traversing Method

The traversing method consists of two activities, i.e., dual programming activity and verification/evaluation activity. In the dual programming activity, a design description text is constructed and a collection of traverser descriptions is added to the text. In the verification/evaluation activity, the function of the design description text is verified and its performance is evaluated with the aid of a collection of traverser descriptions.

The lower half of Fig.III.2.1 shows the traversing method which makes it possible for a software designer to construct the description of processing entities and processed entities and to verify and evaluate the description. The SDES verification/evaluation system consists of a preprocessor, a comiler, etc..

Section III.3
Dual Programming Activity

III.3.1 Traverser: its Path and Attribute

The dual programming activity in the first half of the traversing method means an activity in which a design description text is constructed and a collection of traverser descriptions is added to the text by a user.

A traverser description defines a traverser path, and interactions between a traverser and variables/resources.

A traverser path is a path on which a processed entity proceeds through the system represented by the design description text. A traverser path is specified in terms of a generation point, transfer points and a termination point of a given traverser. The generation point and the termination point may be arbitrarily selected in the text. The examples are an entry point and an exit point of a transaction inputted into an online system, a generation point and a termination point of a message which is passed through two or more processes, and so on. In the case that a traverser corresponds to a processed entity such as a message, the transfer point of the traverser may be specified at the sending point of the processed entity.

The interactions between a traverser and variables/resources are specified in order to associate a collection of variables/resources with the traverser. For each traverser, one or more traverser attributes are defined which represent states of resources or values of variables in the system. A collection of

traverser attributes of a traverser means a collection of states of resources and values of variables which are associated with the behavior of the corresponding processed entity.

A relational expression may be specified on a collection of traverser attributes of each traverser. A relational expression provides an assertion for verifying states of the system in runtime.

The interactions between a traverser and shared variables/resources are specified in order to identify the region of mutually exclusive access for them. The region of mutually exclusive access for a shared variable or resource may be specified on a traverser path. This region is usually called a "critical region" [Hansen 1973], where processes referring the same variable or resource exclude one another. At most one process at a time can be inside this region. The critical region is realized by P- and V-operations for a semaphore variable in [Dijkstra 1965] or ENQ(ENter Queue)- and DEQ(DEpart from Queue)- macros in an assembly language. The most elaborated concept of a critical region is "monitor" [Hoare 1974] [Hansen 1977a 1977b]. In order to verify that a shared variable or resource is accessed exclusively by processes, the user specifies the critical region for a shared variable or resource on a traverser path.

The user may specify the interactions between a traverser and resources in order to collect statistics about elapsed time of a traverser or utilization of resources on the traverser path. We may regard a resource in the original description as a special entity such as a facility entity, a storage entity or a queue entity like in GPSS. A facility entity represents a time-shared resource such as a CPU. A storage entity represents a space-shared resource such as a memory unit. A queue entity represents an ordered set of traversers which are waiting to use a storage entity or a facility entity.

The dual programming activity consists of the following five steps.

(Step-1) Construct a design description text in PL/I. It corresponds to the viewpoint of processing entities, i.e., concurrent processes. This text is an arbitrary detail of executable text.

(Step-2) Specify the realm where the user wants to perform the verification/evaluation in the text. The realm is called a "verification/evaluation realm".

(Step-3) Enumerate a collection of processed entities which proceeds through the verification/evaluation realm. Assign a traverser to each processed entity, and specify its traverser path.

(Step-4) Specify the interaction between a traverser and variabls/resources: specify a collection of traverser attributes and their relational expressions of each traverser, and specify critical regions for shared variables or resources on each traverser path.

(Step-5) Specify paths on which traversers utilize a resource such as a facility, a storage or a queue.

A software system which is constructed in the dual programming activity is executed to verify its function and evaluates its performance by SDES evaluation/verification system. If results are good, the user may go back to step-1 and construct more detailed description, or he may go back to step-2 and modify the verification/evaluation realm. If results are not good, he must go back to step-1 and modify the description. (See Fig.III. 3.1.)

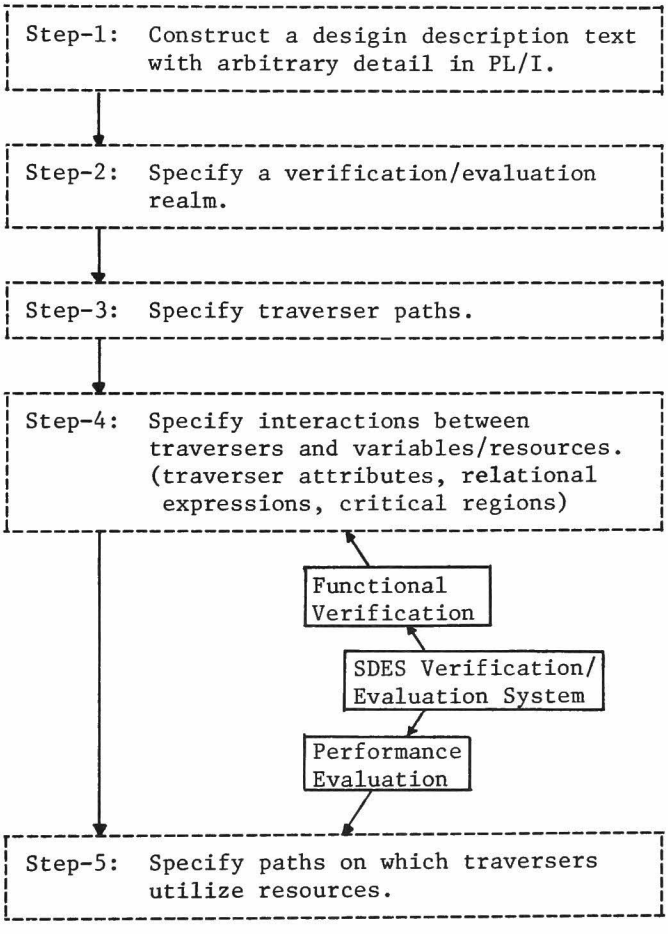


Fig.III.3.1 Steps in Dual Programming Activity

III.3.2 Principles for Specifying Traverser Path

We consider several principles for specifying traverser paths.

(A) Process Activation

(A-1) Transfer of a traverser:
Where a process activates another process in order to process a processed entity, a traverser transfers from the activating process to the activated process. (See Fig.III.3.2.(a).)

(A-2) Split of a traverser:
Where a process activates more than one process in order to process a processed entity, a traverser is split into several traversers, the number of which is equal to the number of the activated processes. Each traverser transfers from the activating process to the activated process. (See Fig.III.3..2.(b).)

(A-3) Merge of traversers:
Where a process starts or restarts to execute its program on condition that it is activated by given several processes, all traversers from these processes are merged into one traverser. (See Fig.III.3.2.(c).)

(B) Interprocess Communication

We consider the situation in which a process sends a message to another process. There are the following two cases. In the first case, a traverser transfers from the sending process to the receiving process. (See Fig.III.3.2.(d).) In the second case, two traversers exist in the sending process and the receiving process, respectively. A traverser at the sending process is splitted when the sending process sends a message. One of the splitted traversers is merged with a traverser at the receiving process. (See Fig.III.3.2.(e).)

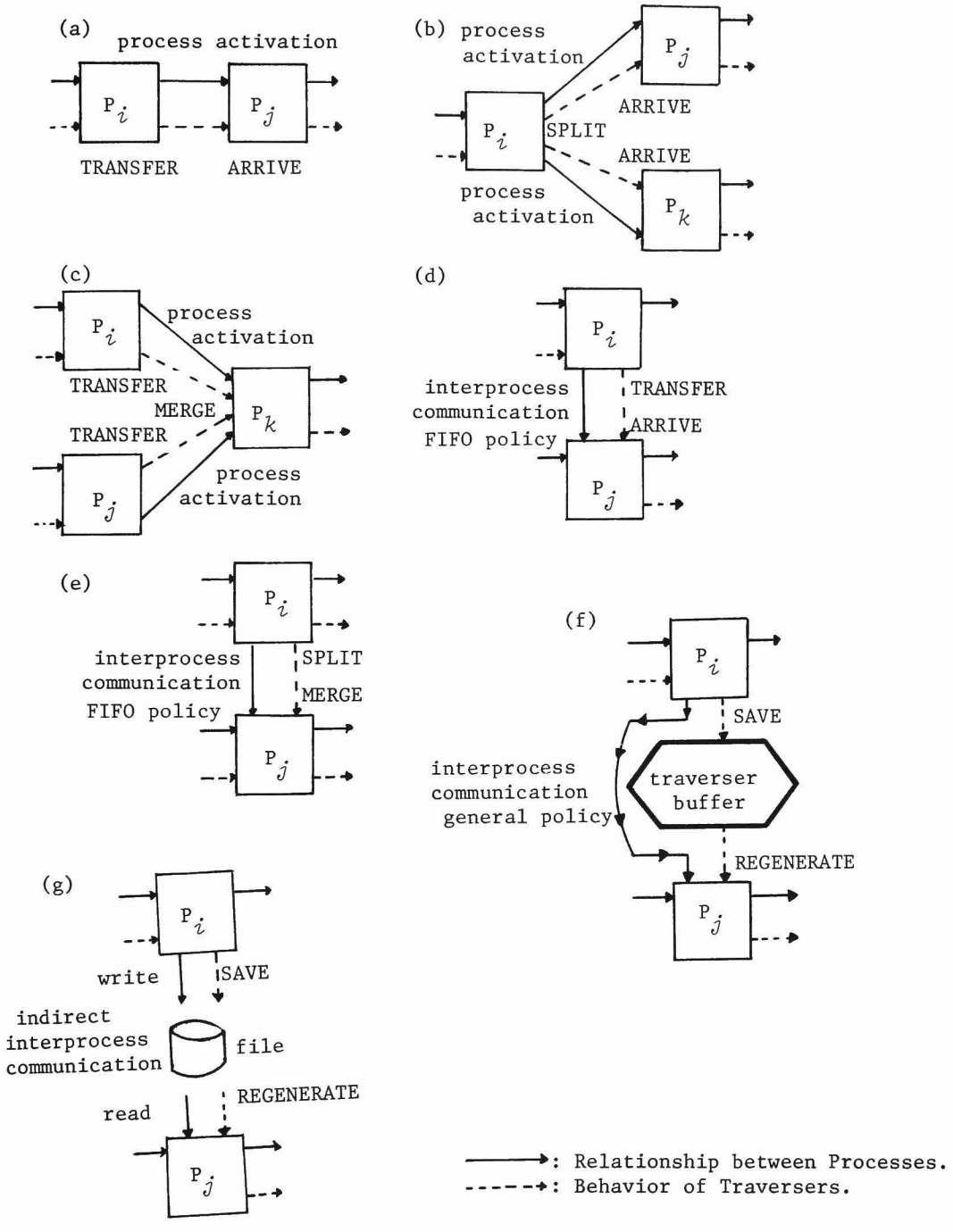


Fig.III.3.2 Principles in Specifying Traverser Paths

These two cases show the situation in which a message receive the First-In First-Out (FIFO) policy. In the situation where a general policy is adopted, a message is received in the order to be associated with the content of the message. When a process sends a message to another process, a traverser information is stored together with the content of the message in a buffer area provided by SDES. This buffer area is named a "traverser buffer ". When a process receives a message, the content of the message coincides with one recorded in the traverser buffer and then a traverser is regenerated. (See Fig.III.3.2.(f).)

(C) Indirect Interprocess Communication via file I/O

A process may communicate with another process via a file record using a write/read operation for a file. A special field is added to a file record for the purpose of recording a traverser information. When a process writes a record into a file, a traverser information is stored in the special field of the record. When a process reads the record from the file, a traverser information is restored and a traverser is regenerated according to this information. (See Fig.III.3.2.(g).)

III.3.3 SDES Language

SDES language includes PL/I and traverser language. The former is used to construct a design description text. The latter is used to construct a collection of traverser descriptions and it consists of traverser statements.

An evaluation text is constructed by adding a collection of traverser statements prefixed by "$" to the original design description text. A traverser statement may hold a label. There may be identifiers of traversers, labels, and identifiers of storages, facilities, queues, variables, or resources. Traverser identifiers are specified in the operand of a traverser statement. A traverser identifier in the operand means a name of the processed entity. When a process executes its program, the process is considered to accompany some traverser, for the process is processing the corresponding processed entity according to its program. A process executes a statement of the original description or a traverser statement of traverser description one by one. When a traverser statement is about to be executed, whether one or more traversers being accompanied with the process are designated in the operand of the traverser statement is examined. If so, the traverser statement is executed.

Traverser statements are classified into four types.

(1) Path statement to define a traverser path.

(2) Interaction statement to define an interaction between a traverser and a variable/resource, and to be used in order to verify the function of the original description.

(3) Utilization statement to define a path on which a traverser utilizes a resource, and to be used to evaluate the performance of the original description.

(4) Case statement to be used to specify a traverser path designated by a value of a variable or a state of a resource. The operand of the case statement allows to specify not only a single value of a variable but also a range of values of a variable.

Path statement includes GENERATE, TERMINATE, TRANSFER, ARRIVE, SPLIT, ASSEMBLE, MERGE, SAVE, REGENERATE, SAVEF and REGENERATF statements.

A GENEREATE statement and a TERMINATE statement specifies the creation and the destruction of a traverser, respectively. A traverser is alive on the path from the point specified by a GENERATE statement to the point specified by a TERMINATE statement.

A TRANSFER statement specifies that a traverser transfers from one process to another. An ARRIVE statement specifies that a traverser arrives at a process. A TRANSFER statement and an ARRIVE statement are used in process activation or interprocess communication between two processes.

A SPLIT statement specifies that one traverser is splitted into two traversers. An ASSEMBLE statement assembles more than one traverser into one traverser. A MERGE statement merges different traversers into one traverser. A SPLIT statement, an ASSEMBLE statement and a MERGE statement are also used in process activation or interprocess communication between one process and two or more processes.

A SAVE statement specifies that a traverser transfers from one process to another process in a general policy of interprocess communication. A REGENERATE statement specifies that a traverser transfers at a process in this interprocess communication. These statements hold an identifier of a traverser and an iden-

tifier of a message as its operand.

A SAVEF statement specifies that traverser information is stored to a special field of a file record. A REGENERATF statement specifies that a traverser is regenerated according to the traverser information. A SAVEF statement and a REGENERATF statement are used in order to verify an indirect interprocess communication via file I/O.

Interaction statement includes ATTACH, DETACH, LOCK, UNLOCK, CHECK, TRACE and TEST statements.

An ATTACH statement specifies that a value of a variable or a state of a resource is attached to a traverser as its traverser attribute. A DETACH statement specifies that a traverser attribute is canceled.

A pair of a LOCK statement and an UNLOCK statement encompasses a critical region for a shared variable or resource.

A CHECK statement specifies that a label at the operand of the CHECK statement is recorded. A TRACE statement specifies that the recorded lablels are outputted.

A TEST statement examines a relational expression for traverser attributes.

Utilization statement includes SEIZE, RELEASE, ENTER, LEAVE, QUEUE, DEPART, MARK, TABULATE and COUNT statements.

A SEIZE statement specifies that a traverser requests to use a facility entity. A RELEASE statement specifies that a traverser finishes to use a facility entity.

An ENTER statement specifies that a traverser requests to enter

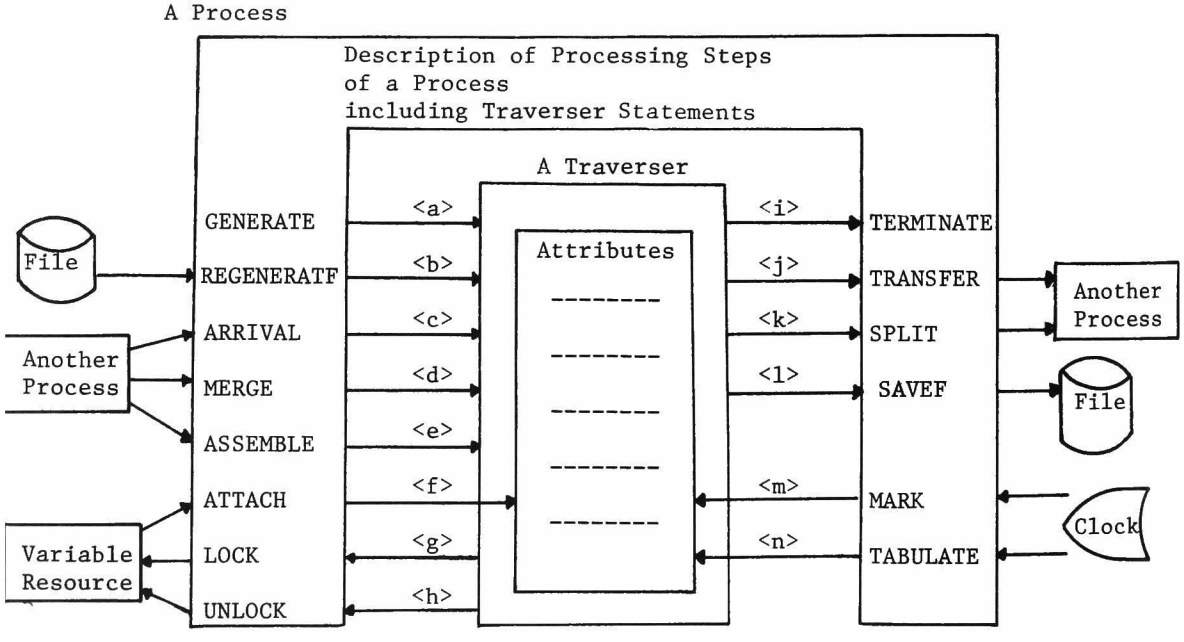


Fig.III.3.3 Relationship between a Process and a Traverser

a storage entity. A LEAVE statement specifies that a traverser leaves a storage entity.

A QUEUE statement specifies that a traverser enters a queue entity to wait until a facility entity or a storage entity is available. A DEPART statement specifies that a traverser departs from a queue entity. Statistics about utilization of a facility entity, a storage entity or a queue entity is automatically collected.

A pair of a MARK statement and a TABULATE statement specifies to collect statistics about elapsed time of a traverser.

A COUNT statement is used for counting the number of traversers which execute this statement.

Appendix 2 shows the syntax of SDES language in terms of Backus Naur Form (BNF).

Fig.III.3.3. represents the relationships between a process and a traverser in terms of traverser statements. An arrow directed to a traverser represents that a traverser starts to proceed through the original description together with a process when a traverser is generated or regenerated (a or b), or when a traverser arrives at the process (c , d or e). An arrow emanated from a traverser represents that a traverser departs from a process when it terminates (i), or when a traverser transfers from a process to another (j or k). An arrow directed to a traverser attribute represents that a value of a variable or a state of a resource is attached to a traverser as its traverser attribute (f) or that elapsed time of a traverser on its path is automatically collected (m and n). An arrow directed to a variable or a resource represents that an critical region for a variable or a resource is specified on a traverser path (g and h).

Section III.4
Verification/Evaluation Acitvity

The verification/evaluation activity in the second half of the Traversing Method is the activity in which SDES verification/evaluation system executes an verification/evaluation text, verifies its function and evaluates its performance with the aid of collections of traverser descriptions. The verification/evaluation activity is divided into two sub-activities:

(1) a functional verification activity by examining traverser paths, traverser attributes or mutually exclusive access for critical regions, and by detecting the occurence of execution errors in a design description text, and

(2) a performance evaluation activity by collecting several statistics. The evaluation activity is automatically performed by our evaluation system with the aid of a collection of traverser descriptions.

In order to examine a traverser path, SDES verification system keeps track of information about traversers accompanied with processes. When a traverser statement is executed, whether a traverser in the operand of the statement exists in the information is examined. If not so, an erroneous behavior of a traverser may occur, and some warning message is issued. This functional verification enables the flow of the corresponding processed entity controlled by one or more processes to be examined.

In order to examine a traverser attribute, SDES verification

system keeps track of information about a collection of traverser attributes. A relational expression in the operand of a TEST statement is examined using the information. This functional verfication enables actions of the corresponding processed entity to the software system and reaction from it to be examined.

Examining mutually exclusive access for a critical region is performed by the use of the information about a traverser which is in the region. While one traverser is in the region, if another traverser tries to enter the region, some warning message is issued. This functional verification enables the existence of simultaneous access for a shared variable and resource to be examined.

SDES verification system performes the verification activity in the above ways where it examines the functions which are associated directly with the user-specified interactions between traversers and variables/resources in the user-specified verification realm. In order to perform more powerful functional verification activity, SDES verification system detects the occurrence of execution errors, identifies their occurrence places and outputs the information (such as traverser attributes and traverser checking points) recorded in the verification activity, whether the execution errors may occur in the inside of the user-specified verification realm or not, or whether they may be associated directly with the user-specified interactions between traversers and variables/resources or not.

In order to perform the performance evaluation, SDES evaluation system keeps track of information about sum of elapsed time of traversers on a path, the number of traversers proceeding on a path, sum of utilization use time of a facility entity, a storage entity or a queue entity, the number of traversers using a facilty entity, storage entity or queue entity, and so on.

This performance evaluation enables performance properties of the software system.

Table III.4.1 shows items of functional verification and performance evaluation by SDES verification/evaluation system.

Functional Verification -- Flows of Processed Entities controlled by one or more Processes -- Action/Reaction of Processed Entities to/from one or more Processes -- Mutually Exclusive Access to Shared Variables by two or more processes -- Other various Execution Errors
Performance Evaluation -- Flow Times of Processed Entities -- Utilization of Resources

Table III.4.1 Items of Functional Verification and Performance Evaluation by SDES

Section III.5
Command System

SDES Command System controls the progress of the dual programming activity and the verification/evaluation activity. In SDES Command System, there are four modes named SDES modes, i.e., dual programming mode, preprocessing mode, compilation mode, linking mode, and execution mode. Fig.III.5.1 shows the transitions between SDES modes and SDES commands which occur the transitions.

In the dual programming mode, a design description text is constructed and collections of traverser descriptions are inserted into the text. There are several subcommands which specify appending, deletion, listing, etc..

In the preprocessing mode, SDES preprocessor preprocesses the text inserted with collections of traverser descriptions and produces a PL/I program. If there is some syntax error in traverser statements, an error message is outputted and the control must be returned to the dual programming mode.

In the compilation mode, the preprocessed text, i.e., a PL/I program, is compiled. If there is some syntax error in a PL/I program, an error message is outputted and the control must be returned to the dual programming mode. When an error message is outputted, it is associated with the syntax error of a PL/I statement in the design description text and it is not associated with a PL/I statement produced by SDES preprocessor. It is desirable that the control must go to the compilation mode after only design description text is constructed in the dual programming mode.

In the linking mode, the compiled program is linked with an editting program for verification/evaluation data.

In the execution mode, the text is executed for verification/ evaluation.

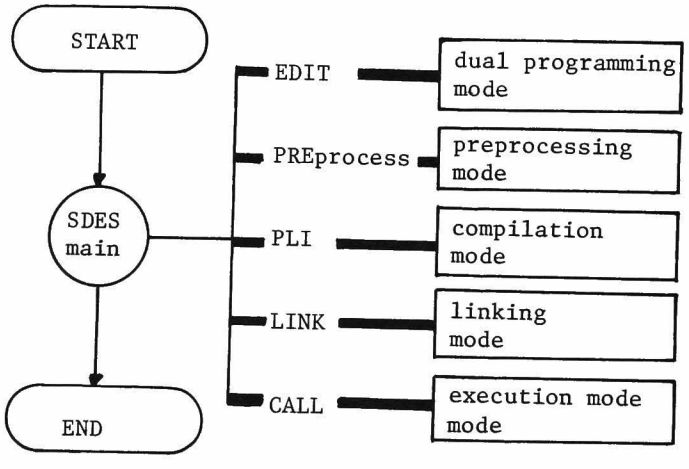


Fig.III.5.1 SDES Modes and Commands

Section III.6
Implementation of SDES

III.6.1 System Description

SDES has been implemented on FACOM M-190 and M-200 (OS IV/F4 TSS) in Data Processing Center of Kyoto University and HITAC M-180 (VOS3 TSS) in Educational Center for Information Processing of Kyoto University. The software organization of SDES is shown in Fig.III.6.1.

The user performs the dual programming activity to construct an evaluation text from his TSS terminal by the use of an editor. A preprocessor adds to the text, tables for verification/evaluation and routines for managing traversers, and it translates each traverser statement into a sequences of calling statements for routines for managing traversers. These tables, routines, and calling statements are written in PL/I. Namely, the output of a preprocessor is a PL/I program. This program is compiled, linked with an editting program for verification/evaluation data, and executed. During the execution, data on the function and performance is recorded on the tables. The editting program references to this data, assembles verification/evaluation results, and displays them on a TSS terminal.

An editor and a PL/I compiler are those provided on M-190 or M-180. A preprocessor and an editting program for verification/evaluation data are newly constructed in PL/I.

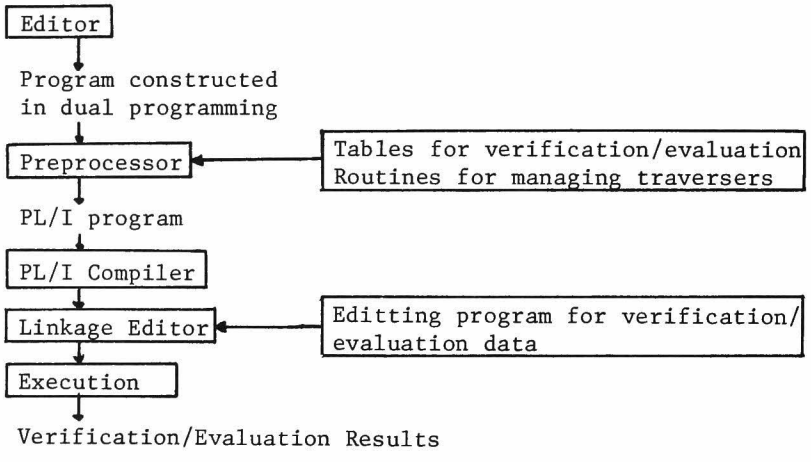


Fig.III.6.1 Software Organization of SDES

III.6.2 Functional Verification Method

Four tables are provided for the management and maintenance of traverser paths, traverser attributes, traverser states and shared variable states. These tables are as follows:

(a) Traverser Table,
(b) Process-to-Traverser table,
(c) Traverser-Queue Table, and
(d) Critical-Region Table.

An entry of (a) consists of an identifier, a generic number, current position, states and atributes of a traverser. The entry is created with an identifier and a generic number of a traverser when a GENERATE statement is executed. The entry is destroyed when a TERMINATE statement is executed. The current position means a position of a traverser in the text. An attribute is defined when an ATTACH statement is executed.

There are three states of a traverser as follows:

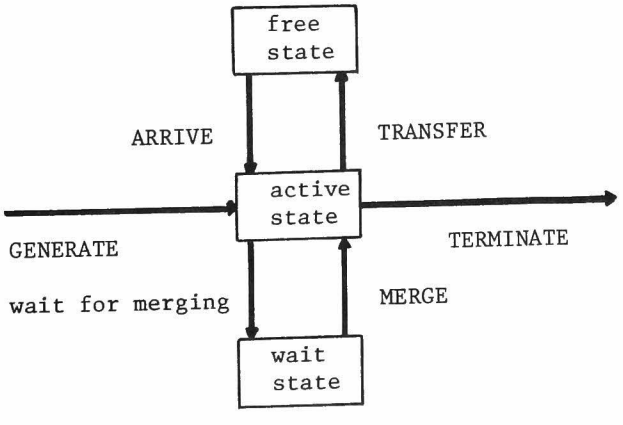


Fig.III.6.2 Traverser State and its Transition

(i) active state: the state in which a traverser is accompanied with a process, i.e., the process performs its processing to a processing entity corresponging to the traverser.

(ii) wait state: the state in which a traverser waits to be merged with another traverser which arrives from a certain process, i.e., a process waits to be synchronized with another process, and

(iii) free state: the stae in which a traverser accompanied by a process leaves and it has not yet arrived at another process.

Fig.III.6.2 shows three states of a traverser and interstate transition. When a traverser executes a GENERATE statement, it becomes to the active state. When a traverser executes a TERMINATE statement in the active state, it goes out in existence. When a traverser executes a TRANSFER statement in the active state, it becomes to the free state. When a traverser starts to be accompanied with a process at the place of an ARRIVE statement, state transition occurs from the free state to the active state. While a traverser waits to be merged with another traverser, it is in the wait state. When a traverser executes a MERGE statement, i.e., it is merged with another traverser, state transition occurs from the wait state to the active state.

An entry of (b) consists of an identifier of a process, an identifier of a traverser and a generic number of a traverser. The entry represents the correspondence between a process and a traverser accompanied by the process. The entry is created when a traverser is created by a GENERATE statement or when a traverser arriving at the place of an ARRIVE statement starts to be accompanied by a process. The entry is maintained while a traverser is in active and wait states. The entry is destroyed when a traverser becomes to free state or when a TERMINATE statement is executed.

An entry of (c) consists of the same components of an entry of (b). The entry represents an entry of a queue where traversers wait to be accompanied by a process.

An entry of (d) consists of an identifier of a shared variable or resource, an identifier of a traverser and its generic number. The entry represents whether the shared variable or resource is used by another traverser. The entry is created when a LOCK statement is executed. It is destroyed when an UNLOCK statement is executed.

In order to examine a traverser path, the correspondence between an identifier of a traverser at the operand of any traverser statement and an identifier of a traverser of the entry in the Process-to-Traverser Table is examined. When a CHECK statement is executed, a label of statement is recorded in an entry of the Traverser Table. When a TRACE statement is executed, a tracing information is editting in terms of data recorded by CHECK statements.

In order to examine attributes of traversers, whether a relational expression of attributes of a TEST statement is satisfied or not is examined by referencing to attributes which are recorded in the Traverser Table.

In order to examine the mutually exclusive access for a variable or resource shared by several concurrent processes, whether an entry of the variable or resource exists or not is examined by referencing to the Critical-Region Table. It means the occurrence of the simultaneous access for the shared variable or resource that an entry has already existed.

In order to detect the occurrence of execution errors whether they may occur in the inside of the user-specified verification realm or not, whether they may be associated directly with the

user-specified interactions between traversers and variables/resources or not, PL/I ON statements are inserted in the preprocessed form of a verification/evaluation text. A condition part of an ON statement designates a type of an execution error, e.g., overflow/undeflow error in computation, protection error for illegal access for a memory, a storage or a device, etc.. An ON statement defines a routine for handling the execution error designated in its condition part. ON statements, which are inserted in the preprocessed form of a text, handles all types of execution errors, so there is a designater "ERROR" in their condition part.

There may be user-specified ON statements in an original text for the purpose of handling a specific execuction error. In order to distinguish the user-specified ON statements with those inserted in the preprocessed form of the text, and to make available the user-specified routine for handling the specific execution error, the scope rule in PL/I language is addopted effectively. A type of an execution error and a routine for handling it in an ON statement holds a scope in the nesting, calling structure of PL/I procedures, i.e., a type of an execution error and a routine in the nearest are available in the nesting structure when a designated execution error is occurred. ON statements inserted in the preprocessed form of a verification/evaluatin text appear in a main procedure which calls the text and in routines for managing traversers. When a user-specified execution error occurs during the execution of the part of the verification/evaluation text, a user-specified routine is invoked for handling the error. When an execution error, except for a user-specified one, occurs, a routine associated in the inserted ON statements is invoked.

Fig.III.6.3 shows the effects of ON statements. In the situation where no ON statement exists in Verification_Evaluation_text, SDES_Error_Handling procedure is invoked when an execution error occurs. When an execution error occurs whose type is different from that designated in an ON statement in Verification_Evaluation_Text, SDES_Error_Handling procedure is invoked. SDES_Error_Handling procedure outputs the occurrence place of an execition error, traverser attributes and traverser checking points.

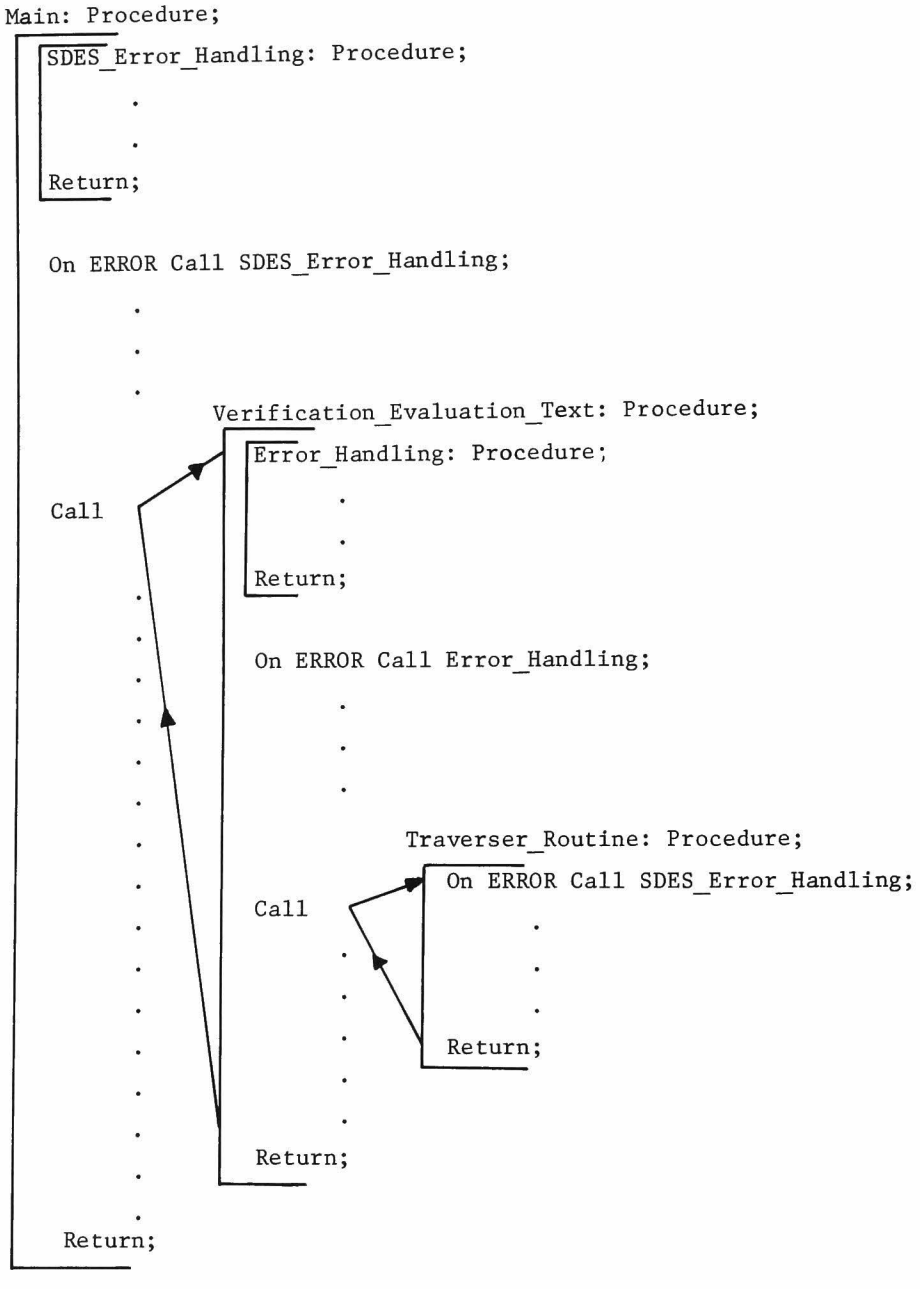


Fig.III.6.3 Effects of ON Statements in Preprocessed Form

III.6.3 Performance Evaluation Method

Two tables are provided in addition to three tables (a), (b) and (c) for collection of statistics on traversers and resources. These tables are as follows:

(e) Traverser-Statistics Table and
(f) Resource-Statistics Table.

When a traverser is created by a GENERATE statement, the clock-time is recorded as a traverser attribute in the Traverser Table. When a traverser destroyed by a TERMINATE statement, the difference between the current clock-time and the clock-time which is recorded in the Traverser Table is recorded in (e). The difference between two clock-times when a traverser encounters a MARK statement and a TABULATE statement is also recorded in (e).

An entry of (e) consists an identifier of a traverser, sum of elapsed time on a path from a GENERATE (or MARK) statement to a TERMINATE (or TABULATE) statement, its maximum, its minimum, and the number of traversers which pass through the path. The entry is updated when the above difference is obtained.

An entry of (f) consists of an identifier of a resource, its type (i.e., facility, storage or queue), the number of traversers which utilize the resource, sum of utilization time, its maximum, and its minimum. The entry is updated when a traverser finishes to utilize a resource, i.e. when a traverser executes a RELEASE, LEAVE or DEPART statement. The clock-time when a traverser starts to utilize a resource is recorded as a attribute in the Traverser Table as well as in the above case.

III.6.4 Routines for Managing Traverser

A pair of a GENERATE statement and a TERMINATE statement is used for the specification of the creation and destruction of a traverser. A pair of a TRANSFER statement and an ARRIVE statement is used for the specification of the transfer of a traverser between two processes. A pair of a MARK statement and a TABULATE statement is used for the specification of the collection of statistics on elapsed time on an arbitrary traverser path. A pair of a SEIZE statement and a RELEASE statement, an ENTER statement and a LEAVE statement, or a QUEUE statement and a DEPART statement is used for the specification of the path of utilizing a facility entity, a storage entity or a queue entity.

Each of these pairs are translated into a sequence of calling statements for three special routines for managing traversers and resources, named a "request routine", an "allow routine" and a "release routine". Other traverser statements are translated into calling statements for routines provided to each.

A path specified by each pair of statements is considered to be a "resource". It is needless to say that a facility entity is a resource whose capacity is one or that a storage entity is a resource whose capacity is greater than or equal to one. A queue entity is considered to be a resource whose capacity is infinite. A path from a GENERATE (or MARK) statement to a TERMINATE (or TABULATE) statement is also considered to be a resource whose capacity is infinite. In addition, the place specified by an ARRIVE statement is considered to be a resource whose capacity is one, because a process starts to accompany only one traverser there.

A request routine manages the request of a traverser to utilize a resource. An allow routine permits a traverser to start to utilize a resource if its current content is less than its capa-

city. It permits a traverser to start to utilize a resource uncondionally if its capacity is infinite. A release routine manages the completion of a traverser to utilize a resource.

A GENERATE statement, a MARK statement, SEIZE statement, or an ENTER statement is translated into a sequence of calling statements for a request routine and an allow routine. A TERMINATE statement, a TABULATE statement, a RELEASE statement, or a LEAVE statement is translated into a calling statement for a release routine. A TRANSFER statement or a QUEUE statement is translated into a calling statement for a request routine, and an ARRIVE statement or a DEPART statement is translated in a sequence of calling statements an allow statement and a release statement.

These three routines refernce to and update the Traverser table, Process-To-Traverser Table, Traverser-Queue Table, Traverser-Statistics Table and Resource-Statistics Table by the use of a special table named "Resource-Management Table". (See Fig.III. 6.4.)

The execution of one traverser statement must not be interleaved by that of another traverser statement, i.e., it must exclusively utilize Resource-Management Table. The preprocessed form of a traverser statement is enclosed by two calling statements realizing critical regions for the table.

Fig.III.6.5 shows examples of preprocessed forms of traverser statements. These are sequences of calling statements for routines for managing traversers and resources, which are enclosed by two calling statements, i.e., "CALL TREGION"s realizing the above critical regions.

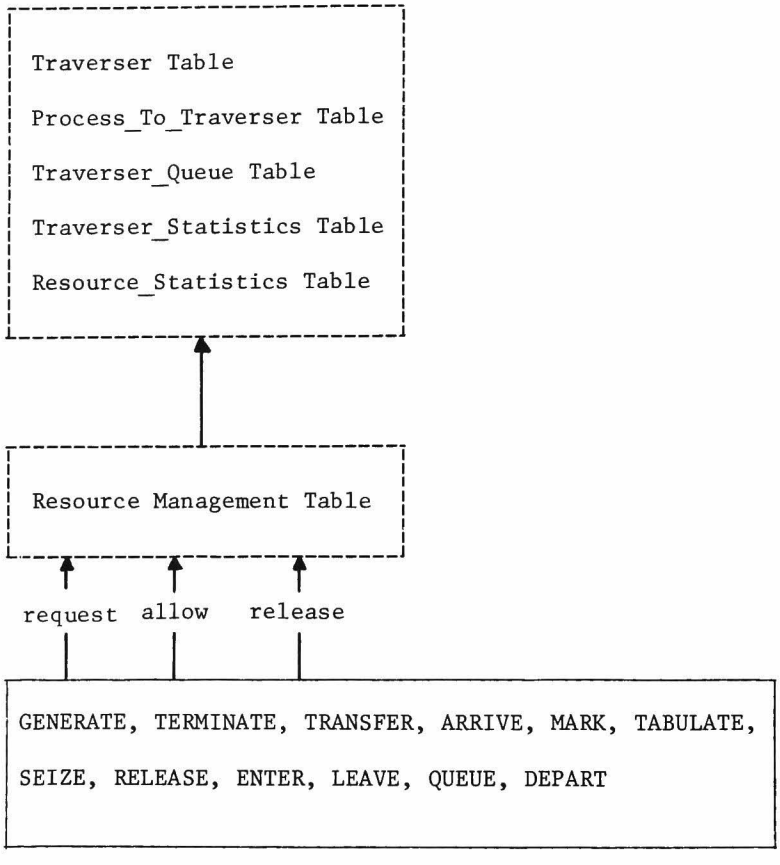


Fig.III.6.4 Resouce Management Table

```
$ SEIZE <traverser id> facility id;

   -->>  CALL TREGION('ENTRY');
         CALL REQUEST(traverser id, facility id);
         CALL ALLOW(facility id);
         CALL TREGION('EXIT);

$ RELEASE <traverser id> facilty id;

   -->>  CALL TREGION('ENTRY');
         CALL RELEASE(traverser id);
         CALL TREGION('EXIT');

$ QUEUE <traverser id> queue id;

   -->>  CALL TREGION('ENTRY');
         CALL REQUEST(traverser id, queue id);
         CALL TREGION('EXIT);

$ DEPART<traverser id> queue id;

   -->>  CALL TREGION('ENTRY');
         CALL ALLOW(queue id);
         CALL RELEASE(traverser id, queue id);
         CALL TREGION('EXIT);

$ TRANSFER <traverser id> entry id;

   -->>  CALL TREGION('ENTRY');
         CALL REQUEST(traverser id, entry id;
         CALL TREGION('EXIT');

$entry id: ARRIVE <traverser id> ;

   -->>  CALL TREGION('ENTRY');
         CALL ALLOW(entry id);
         CALL RELEASE(traverser id, entry id);
         CALL TREGION('EXIT');
```

Fig.III.6.5 Preprocessing of Traverser Descriptions

III.6.5 Editting of Verification/Evaluation Result

During the execution of a verification/evaluation text, an editting program outputs warning messages when a relational expression is not satisfied on traverser attributes, or when the simultaneous access occurs for a shared variable/resource, and it outputs tracing information on a traverser path in the form of lists of checking points when a traverser terminates.

When an execution error occurs, an editting program outputs lists of traverser attributes and checking points about traversers in the verification/evaluation text.

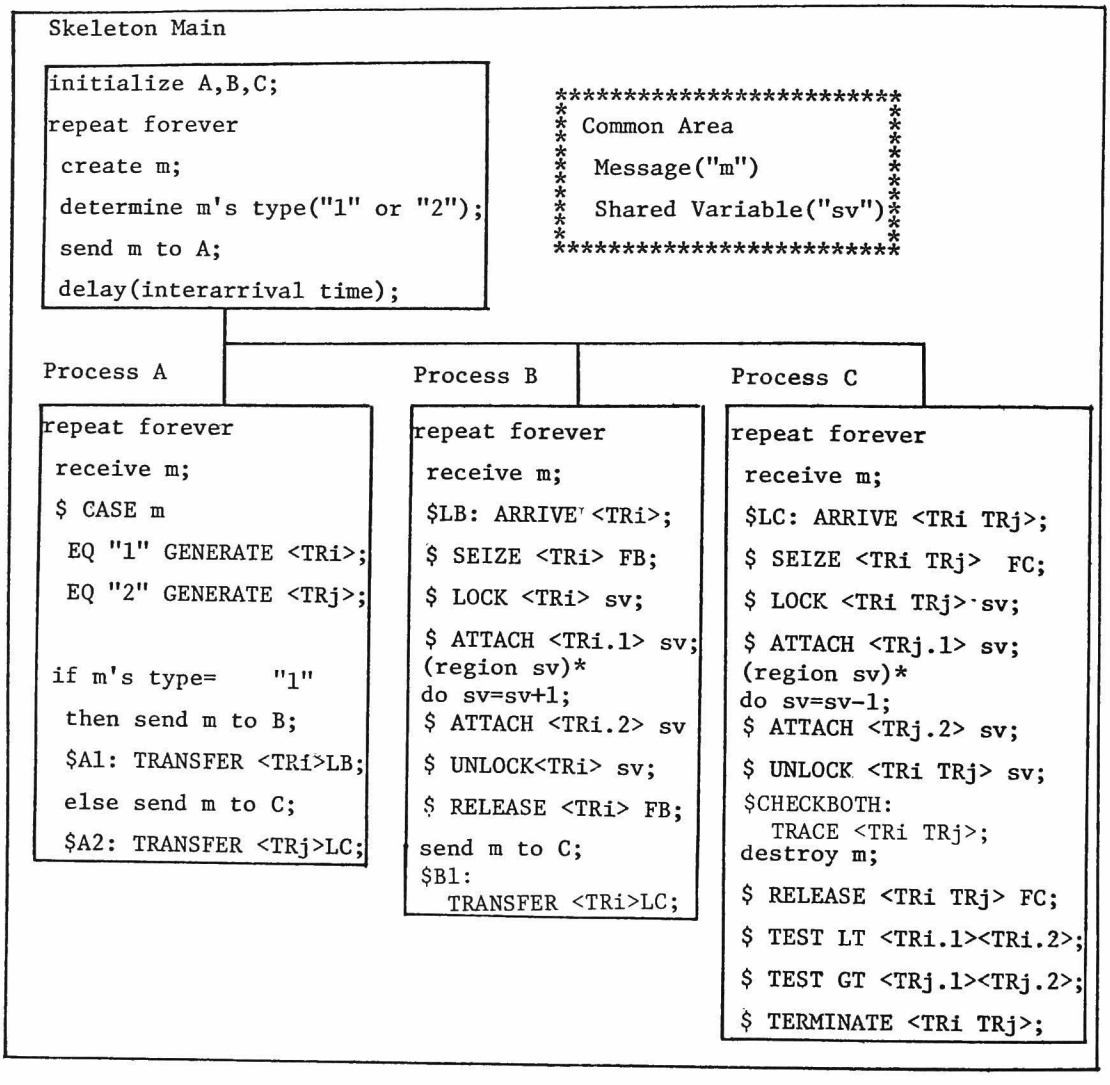
After the execution terminates, an editting program outputs performance information about traversers, facilities, storages and queues, and it outputs lists of traverser attributes and checking points about traversers in the verfication/evaluation text. Performance informaion about traversers is the number of generated traversers, the number of terminated traversers, mean flow time, maximum flow time or minimum flow time. Performance information about facilities or storages is the number of traversers using those, average contents, mean use time, maximum use time, minimum use time and average utilization rate. Performance information about queues is the same as those about facilities or storages execept for average utilization rate.

Section III.7
Examples

Fig.III.7.1 shows the first example of a design description text overlapped with a collection of traverser descriptions. Although the design description text is written in PL/I in the actual environment, the text in Fig.III.7.1 is written in English language flavor (in small letters) for the sake of readability. Traverser descriptions are written in capital letters, and they are prefixed by "$".

There are four processes, that is, "Skeleton Main", "Process A", "Process B" and "Process C". Skeleton Main is a process which simulates the generation of messages, and its priority is higher than that of the other processes. There are two types of messages. A "type-1 message" proceeds through Process A, Process B and Process C. A "type-2 message" proceeds through Process A and process C. Process B and Process C refer a shared variable "sv". Critical regions for "sv" are written in both processes.

A CASE statement in Process A specifies that two types of traversers ("TRi" and "TRj") are generated according to two types of messages, respectively. TRi transfers from Process A to Process B, and TRj transfers from Process A to Process C. In Process B, an ARRIVE statement specifies that TRi arrives at this process, and a TRANSFER statement specifies that TRi transfers from this process to Process C. In Process C, an ARRIVE statement specifies that TRi or TRj arrives at this process, and a TERMINATE statement specifies that TRi or TRj is terminated.



(region sv)* may be missing.

Fig.III.7.1 Example of Message Flow System

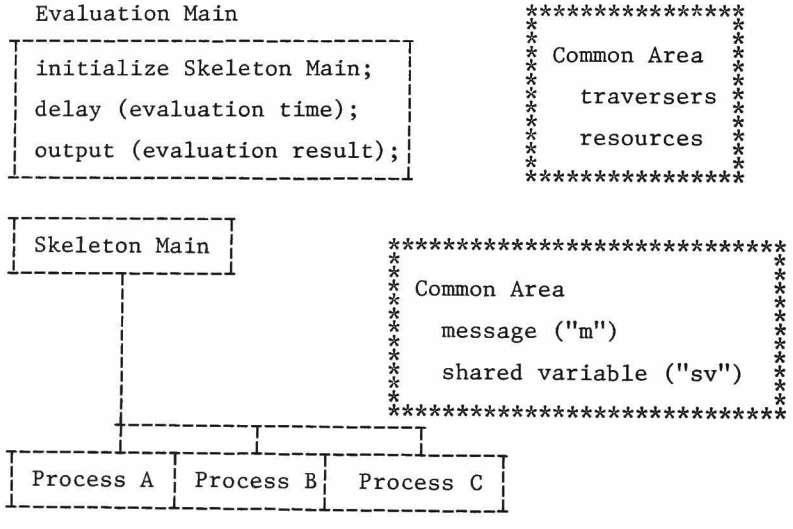
In order to examine the behavior of TRi and TRj, CHECK statements are written in Process A, Process B and Process C, and a TRACE statement is written in Process C. In order to examine the mutually exclusive access for a shared variable "sv", LOCK statements and UNLOCK statements are written for specifying the critical region for "sv" in Process B and Process C. In order to examine a value of a shared variable "sv", ATTACH statements and TEST statements are written. In Process B, an ATTACH statement attaches a value of "sv" to a traverser "TRi" as its first attribute "TRi.1" (its second one "TRi.2") before (after) an assignment statement "sv=sv+1;" is executed. In Process C, an ATTACH statement attaches a value of "sv" to a traverser "TRj" as its first attribute "TRj.1" (its second one "TRj.2") before (after) an assignment statement "sv=sv-1;" is executed. Each of TEST statements in Process C provides a condition which two attributes of TRi or TRj must satisfy.

In order to collect statistics about utilization of Process B and Process C, two facilties ("FB" and "FC") are assigned to them, respectively. SEIZE statements and RELEASE statements are written at the entry points and exit points of Process B and Process C. In order to count the number of traversers passing at the specified point, a CHECK statement is written in Process C.

Fig.III.7.2 shows the preprocessed form of the text, which including a process called "Evaluation Main" and the expanded form of traverser descriptions. Evaluation Main is a process which is added to the design description text for the purpose of the verification/evaluation activity.

The verification and evaluation results are shown in Fig.III.7.3. Fig.III.7.3(a) shows the result of examining traverser paths and examining conditions provided in TEST statements. It shows the histories of traversers and the relations of traverser at-

tributes. Fig.III.7.3(b) shows the result of statistics about elapsed time of TRi and TRj, and utilization of FB and FC, and the result of examining the mutually exclusive access for a shared variable "sv". It shows that mutually exclusive access is performed in safety. Unexpected results in Fig.III.7.3(c) are outputted, because PL/I statements realizing the critical region is missing in the original description. It shows the occurrence of simultaneous access for "sv".



Skeleton Main, Process A, Process B and Process C are preprocessed by Preprocessor.

Fig.III.7.2 Preprocessed Form of Fig.III.7.1

```
¥¥ HISTORY OF TRI            1 ¥¥
  POINT A1          AT 113351858
  POINT LB          AT 113351898
  POINT B1          AT 113351910
  POINT LC          AT 113351953
  POINT CHECKBOTH   AT 113351955
¥¥ (LT TRI.1 TRI.2) IS SATISFIED ¥¥
¥¥ SPECIFIED TRAVERSER TRJ        PASSES AT SPECIFIED POINT CHECKBOTH


¥¥ HISTORY OF TRI           97 ¥¥
  POINT A1          AT 113706488
  POINT LB          AT 113706501
  POINT B1          AT 113706513
  POINT LC          AT 113706533
  POINT CHECKBOTH   AT 113706535
¥¥ (LT TRI.1 TRI.2) IS SATISFIED ¥¥
¥¥ SPECIFIED TRAVERSER TRJ        PASSES AT SPECIFIED POINT CHECKBOTH
¥¥ HISTORY OF TRJ           97 ¥¥
  POINT A2          AT 113707503
  POINT LC          AT 113707510
  POINT CHECKBOTH   AT 113707513
¥¥ (GT TRJ.1 TRJ.2) IS SATISFIED ¥¥
¥¥ SPECIFIED TRAVERSER TRI        PASSES AT SPECIFIED POINT CHECKBOTH
¥¥ HISTORY OF TRI           98 ¥¥
  POINT A1          AT 113708509
  POINT LB          AT 113708525
  POINT B1          AT 113708537
  POINT LC          AT 113708557
  POINT CHECKBOTH   AT 113708560
¥¥ (LT TRI.1 TRI.2) IS SATISFIED ¥¥
¥¥ SPECIFIED TRAVERSER TRJ        PASSES AT SPECIFIED POINT CHECKBOTH
¥¥ HISTORY OF TRJ           98 ¥¥
  POINT A2          AT 113709521
  POINT LC          AT 113709528
  POINT CHECKBOTH   AT 113709533
¥¥ (GT TRJ.1 TRJ.2) IS SATISFIED ¥¥
```

(a) Results of Traverser Paths and Traverser Attributes

Fig.III.7.3 Evaluation/Verification Results of Fig.III.7.1
(to be continued)

```
¥¥ MUTUALLY EXCLUSIVE ACCESS ¥¥ FOR VARIABLE  "SV"
¥¥ IN SAFETY

¥¥ EVALUATION TIME      200114 MS ¥¥
START:113351709 STOP:113711823 TOTAL TIME           200114


¥¥ TRAVERSER STATISTICS ¥¥
    1  TRAVERSER ID.  TRI
GENERATION COUNT              99  TERMINATION COUNT            99
MEAN FLOW            106  MAXIMUM FLOW          230  MINIMUM FLOW   101
    2  TRAVERSER ID.  TRJ
GENERATION COUNT              99  TERMINATION COUNT            99
MEAN FLOW             58  MAXIMUM FLOW          109  MINIMUM FLOW    54

¥¥ FACILITY OR STORAGE STATISTICS ¥¥

    1 ID FB          TYPE FACILITY   CAPACITY      1
     NUMBER ENTRIES          99 AVERAGE CONTENTS          0.0056
     MEAN USE TIME           11
     MAXIMUM USE TIME            60 MINIMUM USE TIME          10
     AVERAGE UTILIZATION          0.0056

 ¥¥ QUEUE STATISTICS IN FRONT OF FB          ¥¥
     NUMBER ENTRIES          99 AVERAGE CONTENTS          0.0004
     MEAN WAIT TIME           0
     MAXIMUM WAIT TIME            4 MINIMUM WAIT TIME           0

 ¥¥ FACILITY OR STORAGE STATISTICS ¥¥

    2 ID FC          TYPE FACILITY   CAPACITY      1
     NUMBER ENTRIES         198 AVERAGE CONTENTS          0.0275
     MEAN USE TIME           27
     MAXIMUM USE TIME            80 MINIMUM USE TIME          25
     AVERAGE UTILIZATION          0.0275

 ¥¥ QUEUE STATISTICS IN FRONT OF FC          ¥¥
     NUMBER ENTRIES         198 AVERAGE CONTENTS          0.0137
     MEAN WAIT TIME           13
     MAXIMUM WAIT TIME            43 MINIMUM WAIT TIME           6

 ¥¥ NUMBER OF TRAVERSERS PASSING AT SPECIFIED POINTS  ¥¥
    99 TRAVERSERS NAMED TRI          PASSED AT SPECIFIED POINT  COUNT1
    99 TRAVERSERS NAMED TRJ          PASSED AT SPECIFIED POINT  COUNT2

 ¥¥ MARK_TABLE STATISTICS ¥¥

    1  MARK_TABLE ID  MARK1
ENTRY COUNT                   99
MEAN FLOW             44  MAXIMUM FLOW           96  MINIMUM FLOW    37

    2  MARK_TABLE ID  MARK2
ENTRY COUNT                    0
```

(b) Results of Performance and Mutual Exclusive Access

Fig.III.7.3 Evaluation/Verification Results of Fig.III.7.1

(to be continued)

```
¥¥ SPECIFIED TRAVÉRSER TRI          PASSES AT SPECIFIED POINT CHECKBOTH
¥¥ HISTORY OF TRI          425 ¥¥
  POINT A1          AT 124229898
  POINT LB          AT 124229917
  POINT B1          AT 124229942
  POINT LC          AT 124229967
  POINT CHECKBOTH   AT 124229969
¥¥ (LT TRI.1 TRI.2) IS SATISFIED ¥¥
¥¥ SPECIFIED TRAVERSER TRJ          PASSES AT SPECIFIED POINT CHECKBOTH
¥¥ SIMULTANEOUS ACCESS FOR "SV"
¥¥ TRAVERSER IN THE REGION
  TRI                    425
¥¥ TRVERSER ATTEMPTING TO ACCESS
  TRJ                    425
¥¥ HISTORY OF TRJ          425 ¥¥
  POINT A2          AT 124230911
  POINT LC          AT 124230920
  POINT CHECKBOTH   AT 124230923
¥¥ (GT TRJ.1 TRJ.2) IS SATISFIED ¥¥
¥¥ SIMULTANEOUS ACCESS FOR "SV"
¥¥ TRAVERSER IN THE REGION
  TRI                    425
¥¥ TRVERSER ATTEMPTING TO ACCESS
  TRI                    426
¥¥ HISTORY OF TRI          426 ¥¥
  POINT A1          AT 124231928
  POINT LB          AT 124231941
  POINT B1          AT 124231974
  POINT LC          AT 124231997
  POINT CHECKBOTH   AT 124232000
¥¥ (LT TRI.1 TRI.2) IS SATISFIED ¥¥
¥¥ SPECIFIED TRAVERSER TRJ          PASSES AT SPECIFIED POINT CHECKBOTH
¥¥ SIMULTANEOUS ACCESS FOR "SV"
¥¥ TRAVERSER IN THE REGION
  TRI                    426
¥¥ TRVERSER ATTEMPTING TO ACCESS
  TRJ                    426
¥¥ HISTORY OF TRJ          426 ¥¥
  POINT A2          AT 124232941
  POINT LC          AT 124232950
  POINT CHECKBOTH   AT 124232952
¥¥ (GT TRJ.1 TRJ.2) IS SATISFIED ¥¥
¥¥ SIMULTANEOUS ACCESS FOR "SV"
¥¥ TRAVERSER IN THE REGION
  TRI                    426
¥¥ TRVERSER ATTEMPTING TO ACCESS
  TRI                    427
```

(c) Occurence of Simultaneous Access

Fig.III.7.3 Evaluation/Verification Results of Fig.III.7.1

Fig.III.7.4 shows the second exeample; it is the system structure of an online sales order entry system which is developed by the use of SDES. This system is a modified one of Widget System in [Yourdan 1972]. The system allows salesmen to input " order transactions", "cancel transactions", "reserve transactions" and "inquiry transactions" from their terminal, warehouse clerks to input "arrival transactions" and "shipment transactions" from their terminals, and accounting clerks to input "charge transactions" and "payment transactions" from their terminals. The current version of SDES is available in TSS environment, so a

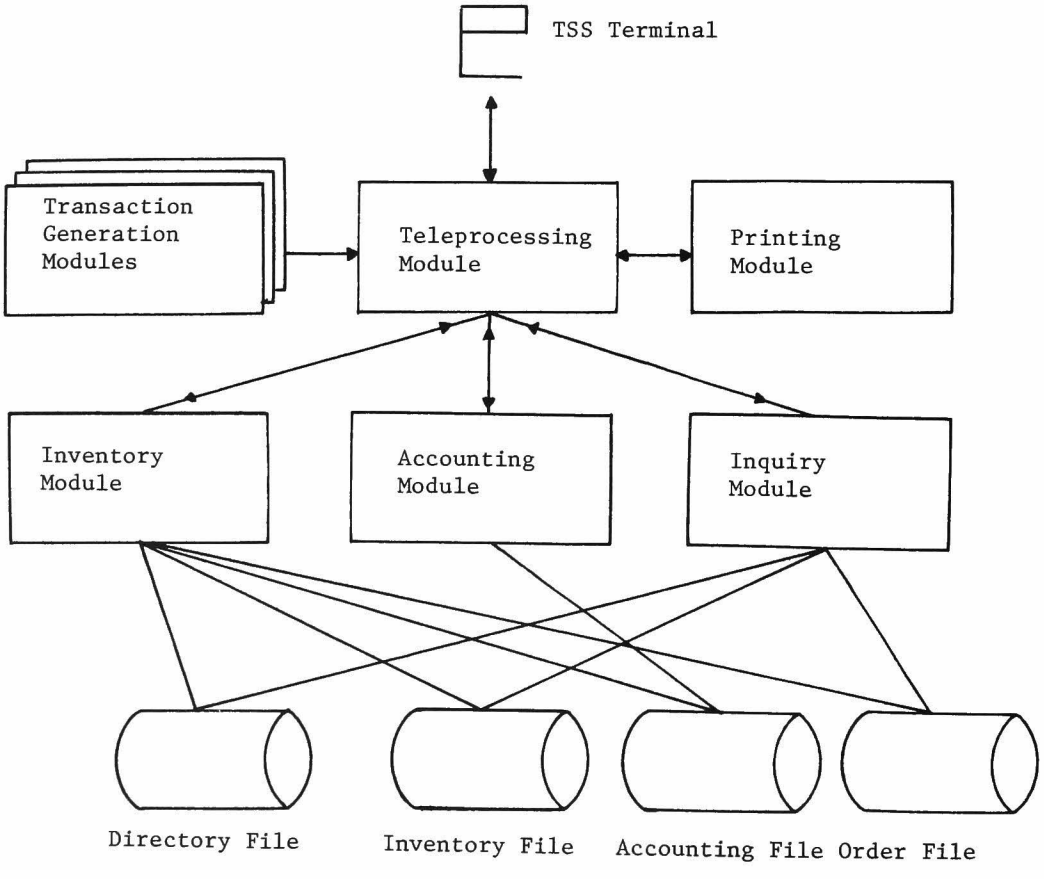


Fig.III.7.4 System Structure of Online Sales Entry System

TSS user may input every type of transactions one by one from his TSS terminal, In order to examine that every type of transactions is processed by this online sales order entry system, "transaction generation modules" are added to the system. Each module in Fig.III.7.4, i.e., "transaction generation modules", "teleprocessing module", "printing module", "inventory module", "accounting module", or "inquiry module", is corresponding to a concurrent process.

Parameters for transactions are as follows.

a) Order number: It is assigned by the system when an order transaction or a cancel transaction is inputted. It is used to identify the order if a salesman later decides to cancel the order, or if he wants to know the status of the order. It is also used in the warehouse to identify the order whether the widgets are shipped.
b) Account number: Each customer is assigned an account number. This number is used to identify each customer.
c) Widget number: It is used to identify a type of a widget and a location in which the widget is kept.
d) Widget amount: It specifies the number of widgets kept in the warehouse.

A brief explanation of each transaction is as follows.

0) Order transaction: It allows a salesman to enter account number, widget number and widget amount. If there are enough widgets to fill the order, it is accepted and a message is sent to a warehouse clerk to instruct the shipment of the widget.
1) Reserve transaction: It has the same parameters and the same effects as an order transaction, except that the warehouse clerk does not receive any message for shipping widgets.
2) Cancel transaction: It allows a salesman to cancel an order

or a quantity of "reserved widget".

3) Arrival transaction: A warehouse clerk uses it to indicate the arrival of fresh widgets from the factory.
4) Shipment transaction: It is used by a warehouse clerk when an order of widgets is being sent to the customer.
5) Payment transaction: it is used by accounting clerks to indicate payments made by customers.
6) Stop-credit transaction: It is used in the event that accounting clerks decided to withdraw the credit of a customer.
7) Restore-credit transaction: It has just the reverse effect of a stop-credit transaction.
8) Charge transaction: It is used to indicate any charges levied against a customer.
9) Credit transaction: It is used to indicate a monetary credit given to a specified customer.
10) Inq-A transaction: It is used by a salesman to determine the quantity of the widgets currently available for sale.
11) Inq-C transaction: It is used by a salesman to determine the status of a customer's account.

An explanation of each module is as follows.

A) Transaction generation module: This module generates transactions according to the user specification.
B) Teleprocessing module: This module interpretes transactions from a TSS terminal and transaction generation modules, and passes them to inventory, accounting and inquiry modules. This module passes reply messages to printing module.
C) Printing module: This muodule outputs reply messages on a TSS terminal.
D) Inventory module: This module processes order, reserve, cancel, arrival and shipment transactions. The module creates dummy charge transactions when shipment transactions are processed.

E) Accounting module: This module processes payment, stop-credit, restore-credit, charge and credit transactions
F) Order module: This module processes inq-A and inq-C transactions.

An explanation of each file is as follows.

i) Inventory file: This file keeps track of the amount of each type of widgets. The amount is increased by cancel or arrival transactions, and it is decreased by order reserve transactions.
ii) Accounting file: This file keeps track of the current amount of money owed by each customer and the status of his credit.
iii) Order file: Order records are created by order or reserve transactions. They are destroyed by shipment or cancel transactions.
iv) Directory file: Keys of order records are chained at each customer in this file.

The online sales order entry system is constructed at three steps.

At the first step, intermodule communications with transactions are described. No parameters of transactions are specified. A traverser is assigned to each transaction. The correct behaviors are verified in terms of TRACE and CHECK statements. (See Fig.III.7.5.)

At the second step, inventory module, order file and directory file are described with complete detail. The correct behaviors are verified in terms of TRACE and CHECK statements. Order transactions create order records in order file. Keys of order records are linked to a chains at each customer in directory file. This process is verified in terms of two ATTACH statements

```
PLEASE ENTER COMMAND:   HOW MANY TRANS IS CREATED?:
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ SPECIFIED TRAVERSER TR4        PASSES AT SPECIFIED POINT SHI1 ¥¥
¥¥ SPECIFIED TRAVERSER TR4        PASSES AT SPECIFIED POINT SHI_CAN
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF ME1              1 ¥¥
  POINT TP1         AT 134150540
  POINT PR1         AT 134150614
TR_NO=14
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF TR0              1 ¥¥
  POINT TP1         AT 134149841
  POINT ST1         AT 134149899
  POINT OR1         AT 134149901
  POINT ST2         AT 134150098
  POINT TP1         AT 134150940
  POINT PR1         AT 134150993
TR_NO= 0
¥¥ SPECIFIED TRAVERSER TR4        PASSES AT SPECIFIED POINT SHI1 ¥¥
¥¥ SPECIFIED TRAVERSER TR4        PASSES AT SPECIFIED POINT SHI_CAN
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF TR5              1 ¥¥
  POINT TP1         AT 134150071
  POINT AC1         AT 134150118
  POINT TP1         AT 134151360
  POINT PR1         AT 134151444
TR_NO= 5
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF TR4              1 ¥¥
  POINT TP1         AT 134149966
  POINT ST1         AT 134150209
  POINT SHI1        AT 134150257
  POINT SHI_CAN     AT 134150344
  POINT ST2         AT 134150443
  POINT TP1         AT 134151787
  POINT PR1         AT 134151873
TR_NO= 4

 TR_NO= 0
 ¥¥ HISTORY OF TR5              5 ¥¥
   POINT TP1         AT 134217460
   POINT AC1         AT 134217501
   POINT TP1         AT 134220930
   POINT PR1         AT 134220946
 TR_NO= 5
 ¥¥ HISTORY OF ME1             35 ¥¥
   POINT TP1         AT 134221022
   POINT PR1         AT 134221039
 TR_NO=14
```

Fig.III.7.5 Evaluation/Verification Results at Step-1

and a TEST statement. When a traverser, which is corresponding to an order transaction, executes the first ATTACH statement, an account number of the order transaction is defined as its first attribute. When it executes the second ATTACH statement, an account number of the established chain in directory file is defined as its second attribute. Two attributes are verified in terms of a TEST statement. (See Fig.III.7.6.) When a chain is not established, two attributes are not identical.

At the third step, all modules and files are described with complete detail. Behaviors and attributes are verified at this step as well as at the second step. Performance Evaluation is performed in terms of SEIZE and RELEASE statements, where facility entities are assigned to modules. (See Fig.III.7.7.)

As SDES is currently implemented under TSS environment of HITAC M-180, FACOM M-190 and FACOM M-200, performance data colected by SDES is affected by the number of active TSS terminals. Fig. III.7.8 shows the relationships between them, where X and Y coordinate axes designate the number of active TSS terminals and performance data of mean flow time of a hundred "TR0"s of the above example, respectively. The SDES verification/evaluation system runs in every TSS sessions. Fig.III.7.8 shows that two quantities have an almost linear relationship, so it is preferable to evaluate performance by using SDES where only one TSS terminal is active.

```
##### MAIN START ######                    ORNO=       1570
PLEASE ENTER COMMAND:   HOW MANY TRANS IS CREATED?:
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF ME1            1 ¥¥
  POINT TP1         AT 141416790
  POINT PR1         AT 141416807
* MESSAGE *   T_NO= 8   TR_NO= 0   BA_NO= 368   MODEL=198
OR_NO=    1571  BA_NO=    368  MODEL=   198  QUANT=          13


¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF TR0            1 ¥¥
  POINT TP1         AT 141416207
  POINT ST1         AT 141416235
  POINT OR1         AT 141416237
  POINT ST2         AT 141417146
  POINT TP1         AT 141417359
  POINT PR1         AT 141417376
* MESSAGE *   T_NO=50   TR_NO= 0   BA_NO= 368   MODEL=198
** ORDER IS ACCEPTED ** OR_NO=    1571


¥¥ SPECIFIED TRAVERSER TR3        PASSES AT SPECIFIED POINT ARR1 ¥¥
¥¥ HISTORY OF TR3            5 ¥¥
  POINT TP1         AT 141426642
  POINT ST1         AT 141449595
  POINT ARR1        AT 141449631
  POINT ST2         AT 141449691
  POINT TP1         AT 141449704
  POINT PR1         AT 141449724
* MESSAGE *   T_NO= 4   TR_NO= 3   BA_NO=   0   MODEL=276
** ARRIVAL IS ACCEPTED **


¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF ME1           12 ¥¥
  POINT TP1         AT 141449883
  POINT PR1         AT 141449903
* MESSAGE *   T_NO= 2   TR_NO= 0   BA_NO=1674   MODEL= 42
OR_NO=    1609  BA_NO=   1674  MODEL=    42  QUANT=          97


¥¥ HISTORY OF TR0           38 ¥¥
  POINT TP1         AT 141427000
  POINT ST1         AT 141449773
  POINT OR1         AT 141449784
  POINT ST2         AT 141450051
  POINT TP1         AT 141450070
  POINT PR1         AT 141450088
* MESSAGE *   T_NO= 0   TR_NO= 0   BA_NO=1674   MODEL= 42
** ORDER IS ACCEPTED ** OR_NO=    1609


¥¥ EQ TR0 . 1 TR0 . 2 IS SATISFIED ¥¥
* MESSAGE *   T_NO=11   TR_NO= 0   BA_NO=   0   MODEL=  0
** ORDER IS ACCEPTED ** OR_NO=    1614
```

Fig.III.7.6 Evaluation/Verification Results of Step-2

```
##### MAIN START ######                          ORNO=       1609
PLEASE ENTER COMMAND:   HOW MANY TRANS IS CREATED?:
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF TR5              1 ¥¥
  POINT TP1         AT 142537069
  POINT AC1         AT 142537083
  POINT TP1         AT 142537472
  POINT PR1         AT 142537489
* MESSAGE *   T_NO=20   TR_NO= 5   BA_NO= 945   MODEL=  0
** PAYMENT IS ACCEPTED **


¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
¥¥ HISTORY OF ME1              2 ¥¥
  POINT TP1         AT 142543259
  POINT PR1         AT 142543276
* MESSAGE *   T_NO= 1   TR_NO= 0   BA_NO= 294   MODEL=151
OR_NO=    1616  BA_NO=    294  MODEL=   151  QUANT=        -20


¥¥ HISTORY OF TR8              1 ¥¥
  POINT TP1         AT 142543097
  POINT AC1         AT 142543111
  POINT TP1         AT 142543683
  POINT PR1         AT 142543825
¥¥ SPECIFIED TRAVERSER TR0        PASSES AT SPECIFIED POINT OR1 ¥¥
* MESSAGE *   T_NO=28   TR_NO= 8   BA_NO= 311   MODEL=  0
** CHARGE IS ACCEPTED **


¥¥ HISTORY OF TR0              7 ¥¥
  POINT TP1         AT 142538620
  POINT ST1         AT 142542981
  POINT OR1         AT 142542987
  POINT ST2         AT 142543702
  POINT TP1         AT 142544256
  POINT PR1         AT 142544315
¥¥ EQ TR0 . 1 TR0 . 2 IS NOT SATISFIED:            294                0
* MESSAGE *   T_NO=55   TR_NO= 0   BA_NO= 294   MODEL=151
** ORDER IS ACCEPTED ** OR_NO=    1616
```

Fig.III.7.7 Evaluation/Verification Results of Step-3

(to be continued)

```
¥¥ EVALUATION TIME      201464 MS ¥¥
START:142519565 STOP:142841029 TOTAL TIME          201464

¥¥ TRAVERSER STATISTICS ¥¥
    1  TRAVERSER ID.  TR0
GENERATION COUNT           38  TERMINATION COUNT          38
MEAN FLOW       13363  MAXIMUM FLOW       19263  MINIMUM FLOW    1371
    2  TRAVERSER ID.  TR1
GENERATION COUNT            0  TERMINATION COUNT           0
    3  TRAVERSER ID.  TR2
GENERATION COUNT            1  TERMINATION COUNT           1
MEAN FLOW       18798  MAXIMUM FLOW       18798  MINIMUM FLOW   18798
    4  TRAVERSER ID.  TR3
GENERATION COUNT            2  TERMINATION COUNT           2
MEAN FLOW       16244  MAXIMUM FLOW       17915  MINIMUM FLOW   14573
    5  TRAVERSER ID.  TR4
GENERATION COUNT            1  TERMINATION COUNT           1
MEAN FLOW       12131  MAXIMUM FLOW       12131  MINIMUM FLOW   12131
    6  TRAVERSER ID.  TR5
GENERATION COUNT            2  TERMINATION COUNT           2
MEAN FLOW         840  MAXIMUM FLOW        1035  MINIMUM FLOW     645
    7  TRAVERSER ID.  TR6
GENERATION COUNT            0  TERMINATION COUNT           0
    8  TRAVERSER ID.  TR7
GENERATION COUNT            0  TERMINATION COUNT           0
    9  TRAVERSER ID.  TR8
GENERATION COUNT            2  TERMINATION COUNT           2
MEAN FLOW        1945  MAXIMUM FLOW        2732  MINIMUM FLOW    1158
   10  TRAVERSER ID.  TR9
GENERATION COUNT            2  TERMINATION COUNT           2
MEAN FLOW        2873  MAXIMUM FLOW        3983  MINIMUM FLOW    1764
   11  TRAVERSER ID.  TR10
GENERATION COUNT            1  TERMINATION COUNT           1
MEAN FLOW        2623  MAXIMUM FLOW        2623  MINIMUM FLOW    2623
   12  TRAVERSER ID.  TR11
GENERATION COUNT            1  TERMINATION COUNT           1
MEAN FLOW        5170  MAXIMUM FLOW        5170  MINIMUM FLOW    5170
   13  TRAVERSER ID.  ME1
GENERATION COUNT            8  TERMINATION COUNT           8
MEAN FLOW         133  MAXIMUM FLOW         410  MINIMUM FLOW      68
   14  TRAVERSER ID.  ME2
GENERATION COUNT            1  TERMINATION COUNT           1
MEAN FLOW          70  MAXIMUM FLOW          70  MINIMUM FLOW      70
   15  TRAVERSER ID.  DUMMY
GENERATION COUNT            1  TERMINATION COUNT           1
MEAN FLOW         506  MAXIMUM FLOW         506  MINIMUM FLOW     506

 ¥¥ QUEUE STATISTICS IN FRONT OF TP1        ¥¥
     NUMBER ENTRIES         109 AVERAGE CONTENTS          0.1578
     MEAN WAIT TIME         291
     MAXIMUM WAIT TIME        1608 MINIMUM WAIT TIME          5

 ¥¥ QUEUE STATISTICS IN FRONT OF ST1        ¥¥
     NUMBER ENTRIES          42 AVERAGE CONTENTS          2.5261
     MEAN WAIT TIME       12117
     MAXIMUM WAIT TIME      17563 MINIMUM WAIT TIME          6
```

Fig.III.7.7 Evaluation/Verification Results of Step-3

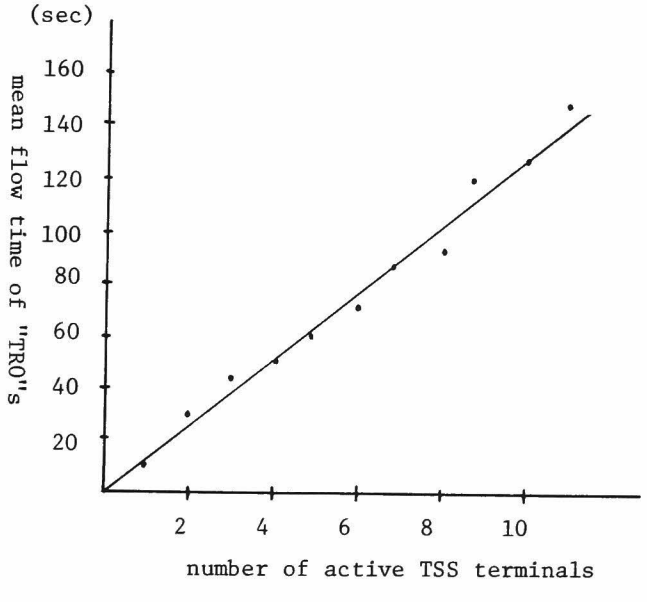


Fig.III.7.8 Statistics under TSS Environment

Section III.8
Conclusion

We have developed the System Description and Evaluation System(SDES) on the large-scale computers HITAC M-180 in the Educational Center for Information Processing of Kyoto University and FACOM M-190 and M-200 in the Data Processing Center of Kyoto University in order to verify the function and evaluate the performance of a software system under design with its detail.

Usually, two activities, i.e., design description activity by programming in a certain programming language (such as PL/I) and simulation activity by rewriting the design description in a certain simulation language (such as GPSS), are performed seperately. SDES integrate these two activities. In our method which is called the Traversing Method, the description of behavior of entities to be processed (called traversers) is overlapped to the original description of processing steps of processing entities (called concurrent processes).

The Traversing Method consists of the following activities:

(1) the dual programming activity in which a collection of traverser descriptions about traverser paths, traverser attributes, and so on is added to the original description by a user, and

(2) the verification/evaluation activity in which the function and performance of the original description is automatically verfied and evaluated by SDES verification/evaluation system with the aid of a collection of traverser descriptions.

Chapter IV
Conclusion

Section IV.1
Summary of the Thesis

Scientific and systematic methodologies are necessary in the design stage of software systems, for the ambiguous, imcomplete or inconsistent design of software systems must not be conveyed to their implementation stage. Even if requirements specifications are valid, no new elaborated implementation methodology will ever produce reliable software systems if the design process proceeds incorrectly. In the design stage, the design must be performed rigourously according to design disciplines and they must be fully verified or evaluated on the aspects of their function and performance.

In this research, two systems are developed for the functional verification and performance evaluation of software systems which are being designed. One of the systems is well suited for functional verification and performance evaluation of software systems which are being designed in top-down fashion. Another system is well suited for functional verification and performance evaluation of software systems which are being designed in their detail.

The first system is called the Interactive Modeling and Simulation System (IMSS). IMSS allows the software designer to

build a hierarchical simulation model in top-down fashion and to execute the hierarchical simulation model at the arbitrary level of modeling process in order to verify it or obtain its statistical properties. In IMSS, the process of the top-down modeling and the simulation execution at the arbitrary level is called "top-down modeling and simulation execution process".

Top-down modeling and simulation execution process conforms well to simulation of software systems which are being developed in top-down fashion. The software designer builds hierarchical simulation models of the object software systems at the same level that the object software systems are being hierarchically developed. He can execute the simulation models at the arbitrary level of the modeling process, i.e., at the arbitrary level of developing process of the object systems, in order to verify the models or obtain their statistical properties. The software designer can verify the functions of software systems and evaluate their performance by building their simulation models and executing them at the same level of designing the software systems. IMSS is a useful support system for the top-down development of software systems.

A software designer can build hierarchical simulation models as he reviews the design of the object software systems. There are chances of finding design errors during design review.

The software designer can execute the simulation at the arbitrary level of his modeling process using "simulation stubs". Already-desined modules corresponds to already-implemented activities. Yet-to-be-designed modules corresponds to yet-to-be-implemented activities. The yet-to-be-implemented activities are replaced by simulation stubs, in which main flows of transactions and communication to their upper activities are described. The execution of a simulation model using simulation stubs provides a software designer with assurance about valid perform-

ance of the object software systems being developed.

The second system is called the System Description and Evaluation System (SDES). SDES is applicable to functional verification and performance evaluation for software systems whose detailed design are being advanced. The detailed design of software sytems means that the software designer determines the detailed individual behavior of software modules and intermodule communication or interaction after he designes the overall structure of the software systems. Especially in the case where software systems consists of more than one concurrent processing module, the software systems must be designed so validly that time-dependent misbehavior of software modules and erroneous intermodule communication can be avoided. In addition, the software systems must be designed with the confidence about valid performance. SDES provides a software designer with a functional verification and performance evaluation tool for software systems whose details are being designed before its implementation.

Each of concurrent processing modules is called a concurrent process in SDES. The detailed design of a software system originates in the description activity about every communication between concurrent processes and main control structure in each process. The original design is interactively refined in stepwise fashion with continual functional verification and performance evaluation for it by SDES.

In order to perform functional verification and performance evaluation of software system whose detail is being designed, the software designer may construct a simple description for verfication and evaluation. This description has the dual relation to the design description. This description is one about the behavior of entities to be processed in a software system.

There are two complementary types of entities in software systems. The first type is a "processing entity". The second type is an entity to be processed, named a "processed entity". Processing entities are concurrent processes. Software systems are designed and implemented from the viewpoint of processing entities. Processed entities are messages, input data, or transactions. In SDES, a description for verification and evaluation is constructed from the viewpoint of processed entities. Processed entities are named "traversers", whose meaning is that processed entities traverse the design description of a software system for its functional verification and performance evaluation. A software designer constructs two dual descriptions about a software system.

When the software designer constructs a description for verification and evaluation, named "traverser description", he reviews the design description. There exists the opportunity of finding design errors in the progress of design review.

Traverser description is the description about messages, input data or transactions which are actually processed by software systems, so traverser description is well suitable for the performance evaluation of the software systems.

Merged description is executed by the SDES verification/evaluation system. SDES verification/evaluation system executes PL/I statements of the description, verifies the design of software systems and evaluates them with the aid of traverser description.

These two systems are useful support systems by which design errors can be avoided by continual functional verification and performance evaluation for software systems on the progress of its design. Two systems were applied to functional verifica-

tion and performance evaluation of an online software system which was being desingned in top-down fashion and in detail. IMSS has been developed on the large-scale computers FACOM M-190 and M-200 in the Data Processing Center of Kyoto University. SDES has been developed on the large-scale computers HITAC M-180 in the Educational Center for Information Processing of Kyoto University and FACOM M-190 and M-200 in the Data Processing Center of Kyoto University.

Section IV.2
Areas for Future Work

We hope to research into more effective design methodologies with continual supports for functional verification and performance evaluation. We try to construct the framework of a new methodology named "two-stage desigining". At the first stage of the two-stage designing, the design of a software system originates in that a software designer constructs the description about the behaviors of entities to be processed. This description is alike a simulation program written in a transaction-oriented simulation language. This description is verified and evaluated automatically on the aspects of its function and performace with the aid of computers. Functional verification facilities must be incorporated into a traditional systems simulator. At the second stage, a software designer constructs the description about processing steps of processing entities and communications between them in stepwise fashion as he reviews the description about entities to be processed. This description is executed by SDES verification/evaluation system. During execution, SDES verification/evaluation system verifies and evaluates this description on the aspects of its fuction and performance with the aid of the descriptopn about entities to be processed. A system based on the two-stage designing is considered to be one which integrates IMSS and SDES and to be widely applicable to the design of software systems.

References

Bell, T.E., Bixler, D.C. and Dyer, M.E.[1977]: An Extendable Approach to Computer-Aided Software Requirements Engineering, IEEE Trans. on S.E., Vol.SE-3, No.1, 49-60, January

Belford, P.C. and Tayler, D.S.[1976]: Specification Verification -- A Key to Improving Software Reliability --, Proc. of Computer Software Engineering, 83-96.

Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R.[1972]: Structured Programming, Academic Press, London.

Dijkstra, E.W.[1965]: Solution of a Problem in Concurrent Programming Control, CACM, Vol.8, No.9, 569-572, September.

Dijkstra, E.W.[1968]: Cooperating Sequential Process, Programming Language, Academic Press, London.

Emshoff, J.R. and Sisson, R.L.[1970]: Design and Use of Computer Simulation Models, The Macmillan Company, New-York.

Good, D.I., London, R.L. and Bledsoe, W.W.[1975]: An Interactive Program Verification System, IEEE Trans. on S.E., Vol.SE-1, No.1, 59-67, March.

Graham, R.M., Clancy, G.J. and DeVaney, D.B.[1973]: A Software Design and Evaluation System, CACM, Vol.16, No.2, 110-116, February.

Greenberger, M. and Jones, M.M.[1965]: On-line Computation and Simulation: The OPS-3 System, M.I.T.Press, Massachusetts,

Greenberger, M. and Jones, M.M.[1966]: Online Simulation in the OPS System, Proc. of the 21st National Conference, ACM, 131-138.

Greenberger, M. and Jones, M.M.[1967]: Online Incremental Simulation, Proc. of the IFIP Working Conference on Simulation Programming Language, Oslo.

Hansen, P.B.[1970]: The Nucles of a Multiprogramming System, CACM, Vol.13, No.4, 238-250, April.

Hansen, P.B.[1972]: Structured Multi Programming, CACM, Vol.15, No.7, 574-577, July.

Hansen, P.B.[1973]: Operating System Principles, Prentice-Hall, London.

Hansen, P.B.[1974]: A Programming Methodology for Operating System Design, Proc. of IFIP 74, 394-397.

Hansen, P.B.[1977a]: The Architecture of Concurrent Programs Prentice-Hall, London.

Hansen, P.B.[1977b]: Experience with Modular Concurrent Programming, IEEE Trans. on S.E., Vol.SE-3, No.2, 156-159, March.

Hoare, C.A.R.[1974]: Monitors: An Operating System Structuring Concepts, CACM, Vol.17, No.10, October.

Howard, J.H. and Alexander, W.P.[1973]: Analyzing Sequences of Operations Performed by Programs, in Program Test Method, Prentice-Hall.

IBM: General Purpose Systems Simulator/360, User's Manual, IBM

Ichbiah, J.D. and Morse, S.P.[1972]: General Concepts of the Simula 67, Programming Language, Annual Review in Automatic Programming, Vol.7, 65-94.

Itoh, K., Kubo, M., Tabata, K. and Ohno, Y.[1975]: Top-Down Modeling and Simulation system-GMSS Simulator-, Proc. of the 16-th Annual Convention of IPSJ, 419-420, November, in Japanese.

Itoh, K.[1976]: Online Simulation System GMSS, Master's Thesis, Kyoto University, February.

Itoh, K., Agusa, K., Kubo, M., Tabata, K. and Ohno, Y.[1977a]: Several Approaches to Performance Evaluation for Software, Meeting Memo of System Performance Evaluation of IPSJ, SE18-4, April, in Japanese.

Itoh, K., Tabata, K. and Ohno, Y.[1977b]: An Evaluation Methodology of Function and Performance of Concurrent Process System, Proc. of the 18-the Annual Convention of IPSJ, 637-638, Octover, in Japanese.

Itoh, K.[1977c]: An Evaluation System For Concurrent Processes by the Traversing Method, Meeting Memo of Dept. of Information Science, Kyoto University, December, in Japanese.

Itoh, K., Tabata, K. and Ohno, Y.[1978a]: An Evaluation System of Concurrent Processes by the Traversing Method, Proc. of 3rd USA-Japan Computer Conference, 41-45, October.

Itoh, K., Kubo, M., Tabata, K. and Ohno, Y.[1978b]: Interactive Modeling and Simulation System, Proc. of International Conference on Cybernetics and Society, 1247-1252, November.

Itoh, K., Tabata, K. and Ohno, Y.[1979a]: System Description and Evaluation System: SDES, Trans. of IPSJ, Vol.20, No.4, 355-362, July, in Japanese.

Itoh, K., Nagai, T.,Tabata, K. and Ohno, Y.[1979b]: Software Description of SDES and its application, Proc. of the 20-th Annual Convention of IPSJ, 337-338, July, in Japanese.

Kubo, M., Itoh, K., Tabata, K. and Ohno, Y.[1975]: Top-Down Modeling and Simulation System-the availability of Graphics-, Proc. of the 16-th Annual Convention of IPSJ, 265-266, November, in Japanese.

Kubo, M., Itoh, K., Tabata, K. and Ohno, Y.[1976]: Online Simulator of GMSS, Meeting Memo of Man-Machine System of IPSJ, MMS24-1, March, in Japanese.

Liskov, B.H. and Zilles, S.N.[1974]: Programming with Abstract Data Types, SIGPLAN Notices, Vol.9, No.4, 50-59, April.

Markowitz, H.M., Hausner, B. and Karr, H.W.[1963]: SIMSCRIPT A Simulation Programming Language, Prentice-Hall, London.

Masaki, T., Itoh, K., Tabata, K. and Ohno, Y.[1978]: Design and Implementation of VDL Interpreter by LISP, Proc. of the 19-th Annual Convention of IPSJ, 119-120, August, in Japanese.

Mills, H.[1971]: Top down Programming in large Systems, Debugging techniques in Large systems by R.Rustin, Prentice-Hall, London, 41-55.

Myers, G.J.[1975]: Reliable Software through Composite Design, Mason/Charter Publishers, New York.

Nakanishi, T.[1969] : System Simulator, Sangyo Tosho, Tokyo, in Japanese.

Ohno, Y., Tabata, K., Agusa, K., Kubo, M. and Itoh, K.[1976]: Graphical Modeling and Simulation System Reference Manual, Laboratory of Professor Ohno, Department of Information Science, Kyoto University, Kyoto, March.

Parnas, D.L.[1971]: Information Aspects of Design Methodologies, Proc. of IFIP-71, 339-344.

Ramamoorthy, C.V.[1966]: Analysis of Graphs by Connectivity Considerations, JACM, Vol.13, 211-222, April.

Rose. C.W.[1972]: LOGOS and the Software Engineer, Proc. of FJCC, 311-323.

Ross, D.T. and Schoman, H.E.Jr[1977]: Structured Analysis for Requirements Definition, IEEE Trans. on S.E., Vol.SE-3, No.1, 6-15, January.

Schechter, D.[1978]: The Skeleton Programming Methodology, Datmation, 147-150, November.

Snowden, R.A.[1972]: PEARL: An Interactive System for the Preparation and Validation of Structured Programs, SIGPLAN Notices, Vol7, No.3, 3-15, March.

Sugimoto, S., Itoh, K., Tabata, K. and Ohno, Y.[1978]: Software Design and Evaluation System, Proc. of the 19-th Annual Convention of IPSJ, 339-340, August, in Japanese.

Sugimoto, S., Itoh, K., Tabata, K. and Ohno, Y.[1979]: Operational Definition Method for Concurrent Programming Language, Proc. of the 20-th Annual Convention of IPSJ, 163-164, July, in Japanese.

Tabata, K., Wada, Y. and Ohno, Y.[1975]: Top-Down Modeling and Simulation with Graphics, Proceedings of 2nd USA-JAPAN Computer Conference, 410-415, August.

Tabata, K., Itoh, K., Hirota., T. and Ohno, Y.[1978]: Tools for Software Development, Japan IBM Intelligent Programming System Symposium, November.

Tabata, K., Sugimoto, S., Masaki, T., Itoh, K. and Ohno, Y. [1979]: Concurrent Lisp, Proc. of the 20-th Annual Convention of IPSJ, 181-182, July, in Japanese.

Teichroew, D. and Hershey, E.A.[1977]: PSL/PSA: A Computer-Aided Structured Documentation and Analysis of Information Processing Systems, IEEE Trans. on S.E., Vol.SE-3, No.1, 41-48 January.

Walde, W.A., Eig, D. and Hunter, S.R.[1968]: GPSS/360-Norden, an Improved System Analysis Tool, IEEE Trans. Systems Science and Cybernetics, Vol.SSC-4, No.4, 442-445, November.

Wirth, N.[1971a]: Program Development by Stepwise Refinement, CACM, Vol.14, No.4, 221-227, April.

Wirth, N.[1971b]: The Programming Language Pascal, Acta Informatica, No.1, 35-63.

Wirth, N.[1977a]: Modula: a Language for Modular Multiprogramming, Software-Practice, and Experience, Vol.7, 3-35, January.

Wirth, N,[1977b]: Toward a Discipline of Real-Time Programming, CACM, Vol.20, No.8, 577-583, August.

Wulf, W.A.[1971]: BLISS: A Language for Systems Programming, CACM, Vol.14, No.12, 780-791, December.

Wulf, W.A., London, R.L. and Shaw, M.[1976]: An Introduction to the Construction and Verification of Alphard Programs, IEEE Trans. on S.E., Vol.SE-2, No.4, 253-265, December.

Yoshizawa, S., Hayakawa, K. and Nakanishi, T.[1975]: Development of Online Simulation System OLSS-1, in Japanese, Seikei University, Tokyo, in Japanese.

Yourdon, E.[1972]: Design of On-Line Computer Systems, Prentice-Hall.

Ziegler, E.W.[1968]: The GPSS On-Line Monitor, IEEE Trans. on Systems Sciences and Cybernetics, Vol.SSC-4, No.4, 438-441, November.

List of Publications and Technical Reports

1. Top-Down Modeling and Simulation system-GMSS Simulator-, Itoh, K., Kubo, M., Tabata, K. and Ohno, Y.[1975], Proc. of the 16-th Annual Convention of IPSJ, November, 419-420, in Japanese.

2. Top-Down Modeling and Simulation System-the availability of Graphics-, Kubo, M., Itoh, K., Tabata, K. and Ohno, Y.[1975] Proc. of the 16-th Annual Convention of IPSJ November, 265-266, in Japanese.

3. Online Simulation System GMSS, Itoh, K.[1976], Master's Thesis, Kyoto University, February.

4. Graphical Modeling and Simulation System Reference Manual, Ohno, Y., Tabata, K., Agusa, K., Kubo, M. and Itoh, K.[1976], Laboratory of Professor Ohno, Department of Information Science, Kyoto University, March.

5. Online Simulator of GMSS, Kubo, M., Itoh, K., Tabata, K. and Ohno, Y.[1976], Meeting Memo of Man-Machine System of IPSJ, MMS24-1, March, in Japanese.

6. Several Approaches to Performance Evaluation for Software, Itoh, K., Agusa, K., Kubo, M., Tabata, K. and Ohno, Y. [1977], Meeting Memo of System Performance Evaluation of IPSJ, SE18-4, April, in Japanese.

7. An Evaluation Methodology of Function and Performance of Concurrent Process System, Itoh, K., Tabata, K. and Ohno, Y. [1977], Proc. of the 18-the Annual Convention of IPSJ, 637-638, Octover, in Japanese.

8. An Evaluation System For Concurrent Processes by the Traversing Method, Itoh, K.[1977], Meeting Memo of Dept. of Information Science, Kyoto University, December, in Japanese.

9. Software Design and Evaluation System, Sugimoto, S., Itoh, K., Tabata, K. and Ohno, Y.[1978]: Proc. of the 19-th Annual Convention of IPSJ, 339-340, August, in Japanese.

10. Design and Implementation of VDL Interpreter by LISP, Masaki, T., Itoh, K., Tabata, K. and Ohno, Y.[1978], Proc. of the 19-th Annual Convention of IPSJ, 119-120, August, in Japanese.

11. An Evaluation System of Concurrent Processes by the Traversing Method, Itoh, K., Tabata, K. and Ohno, Y.[1978], Proc. of 3rd USA-Japan Computer Conference, 41-45, October.

12. Interactive Modeling and Simulation System, Itoh, K., Kubo, M., Tabata, K. and Ohno, Y.[1978], Proc. of 1978 International Conference on Cybernetics and Society, 1247-1252, November.

13. Tools for Software Development, Tabata, K., Itoh, K., Hirota., T. and Ohno, Y.[1978], Japan IBM Intelligent Programming System Symposium, November.

14. System Description and Evaluation System: SDES, Itoh, K., Tabata, K. and Ohno, Y.[1979], Trans. of IPSJ, Vol.20, No.4, 355-362, in Japanese

15. Software Description of SDES and its application, Itoh, K., Nagai, T.,Tabata, K. and Ohno, Y.[1979], Proc. of the 20-th Annual Convention of IPSJ, 337-338, July, in Japanese.

16. Concurrent Lisp, Tabata, K., Sugimoto, S., Masaki, T., Itoh, K. and Ohno, Y.[1979], Proc. of the 20-th Annual Convention of IPSJ, 181-182, July, in Japanese.

17. Operational Definition Method for Concurrent Programming Language,Sugimoto, S., Itoh, K., Tabata, K. and Ohno, Y. [1979], Proc. of the 20-th Annual Convention of IPSJ, 163-164, July, in Japanese.

Appendix 1 Syntax of IMSS Language

We give a formal definition of IMSS language. The meta-language used in this definition is a modification of Backus-Naur Form. Statements are delimited by semicolon(;). To achieve compact definition, we use the following notational devices.

|	read "or" is used to denote a range of options from which one must be taken.
$\{\ \}_i^j$	are used for a representation of repetitive occurrences of the enclosed string where i is the minimum number of repetition required and j is the maximum number of repetition permitted.
$\{\ \}_1$	denote the repetition of the string one or more times.
$\{\ \}_0$	denote the repetition of the string none or more times.

[model description]:=[model heading][model body][model tail]
[model heading]:=IMSS[model identifier-1]([activity list]);
{[model declaration part]}$_0^1$
[model body]:=[model definition part]
[model tail]:=END IMSS;
[model declaration part]:={[dec];}$_1$
[model definition part]:={[activity definition]}$_1${[function definition]}$_0$
[activity list]:=[activity identifier-1]{,[activity identifier-1]}$_0$
[activity definition]:=[activity heading][activity body]
[activity tail]
[activity heading]:=ACTIVITY[activity identifier-1];
{[activity declaration]}$_0^1$
[activity body]:=ENTRY;{[model st];}$_1$EXIT;
[activity tail]:=END[activity identifier-1];
[activity declaration]:={[dec];}$_1$
[function definition]:=[function heading][function body]
[function tail]
[function heading]:=FUNCTION[function identifier-1]{([parameter list])}$_0^1$;{[common dec];}$_0^1$

```
[function body]:={[model st];}1
[function tail]:=END[function identifier-1];
[parameter list]:=[variable identifier-1]{,[variable identifier-1]}0
[dec]:=[facility dec]|[storage dec]|[queue dec]|[semaphore dec]|
       [varible dec]|[actentity dec]|[comact dec]|[initial dec]
[variable dec]:=[common dec]|[private dec]
[declared item-1]:=[identifier-1]([capacity]){([array])}10
                   {-[service-1]}10
[declared item-2]:=[identifier-1]([capacity]){([array])}10
                   {-[service-2]}10
[declared item-3]:=[identifier-1]{([array])}10
[declared item-4]:=[identifier-1]
[array]:=[constant]{,[constant]}10
[service-1]:=FIFO|LIFO|RAND
[service-2]:=FIFO|LIFO|SFPO|RAND
[capacity]:=[constant]
[facility dec]:=FACILITY[declared item-1]{,[declared item-1]}0
[storage dec]:=STORAGE[declared item-2]{,[declared item-2]}0
[queue dec]:=QUEUE[declared item-3]{,[declared item-3]}0
[semaphore dec]:=SEMAPHORE[declared item-3]{,[declared item-3]}0
[common dec] :=COMMON[declared item-3]{,[declared item-3]}0
[private dec]:=PRIVATE[declared item-4]{,[declared item-4]}0
[actentity dec]:=ACTENTITY[declared item-4]{,[declared item-4]}0
[comact dec]:=COMACT[declared item-4]{,[declared item-4]}0
[initial dec]:=INITIAL{[initial com]|[initial sem]}11
              {,[initial com]|[initial sem]}0
[initial com]:=[common variable identifier-1]/[common ele]/
[initial sem]:=[semaphore identifier-1]/[sem ele]/
[com ele]:=[ele]{,[ele]}0|[ele] *[constant]
[sem ele]:=[elt]{,[elt]}0|[elt] *[constant]
[ele]:={-}10[constant]
[elt]:=ON|OFF
```

```
[model st]:=[non structure st]|[structure st]
[non structure st]:=[simple st]|[compound st]|[labeled st]
[simple st]:=[put st]|[get st]|[use st]|[preempt st]|[in st]|
             [set st]|[reset st]|[hold st]|[wait until st]|
             [assign st]|[activity st]
[labeled st]:=[identifier-1]:[non structure st]
[compound st]:=BEGIN;{[simple st];}1END
[put st]:=PUT[arithmetic exp]TO[storage identifier-2]
[get st]:=GET[arithmetic exp]FROM[storage identifier-2]
[use st]:=USE[facility identifier-2]FOR[arithmetic exp]
[preempt st]:=PREEMPT[facility identifier-2]
              FOR[arithmetic exp]
[in st]:=IN[queue identifier-2]
[hold st]:=HOLD[arithmetic exp]
[set st]:=SET[semaphore identifier-2]
[reset st]:=RESET[semaphore identifier-2]
[wait until st]:=WAIT UNTIL[boolean exp]
[assign st]:=[variable identifier-2]=[arithmetic exp]
[activity st]:=ACT[actentity identifier-1]|USE[facility identifier-2
               FOR[actentity identifier-1]
[structure st]:=[if st]|[repeat st]|[do st]|[case st]|[condition st]
                [split st]|[match st]
[if st]:=IF[boolean exp];THEN[compound st]{;ELSE[compound st]}10
[case st]:=CASE[variable identifier-2]OF[constant]
          {;[labeled st]}  constant
                           constant
[condition st]:=CONDITION[constant];{[boolean exp]->
                [compound st];}  constant  T->[compound st]
                                 constant
[repeat st]:=REPEAT[non structure st];UNTIL[boolean exp]
[do st]:=WHILE[boolean exp];DO[compound st]
[split st]:=SPLIT[constant]{;[labeled st]}  constant
                                            constant
[match st]:=MATCH[match identifier-1]
[execution description]:=[execution heading][execution body]
                         [execution tail]
[execution heading]:=EXECUTION[model identifier-1]
[execution body]:={[execution block]}1{[table block]}0
[execution tail]:=END[model identifier-1];
```

```
[execution block]:=[generate st];{[assign st];}0[exec st];
                   [terminate st];{[stop st];}0
[generate st]:=GENERATE([initial clock],[interval time],
               [generate st])
[exec st]:=EXEC[activity identifier-1]
[stop st]:=STOP({[terminate count]}0,{[terminate clock]}0)
[terminate st]:=TERMINATE
[table block]:={[tabulate st];}1{[table definition]}1
[tabulate st]:=TABULATE[variable identifier-2]AT[label identifier]
               IN[table identifier-1]|TABULATE FROM
               [label identifier-1]TO[label identifier-1]IN
               [table identifier-1]
[table definition]:=TABLE[table identifier-1],[lower limit],
                    [upper limit],[step]
[comment]:=${[letter]|[digit]}1$
[identifier-1]:=[letter]{[letter]|[digit]}0^9
[identifier-2]:=[identifier-1]{([subscript])}0
[letter]:=a|b|.....|z|A|B|.....|Z
[digit]:=0|1|2|3|4|5|6|7|8|9
[subscript]:={[variable identifier-1]|[constant]}1^1
            {,[variable identifier-1]|[constant]}0^1
[constant]:={[digit]}1^0
[system variable]:=[storage variable]|[facility variable]|
                   [queue variable]|[semaphore variable]
[arithmetic exp]:=[term]|[adding operator] [term]|
                  [arithmetic exp][adding operator][term]
[term]:=[factor]|[term][multiplying operator][factor]
[factor]:=[primary]|[primary]**[constant]
[primary]:=[constant]|[variable identifier-2]|[function identifier-1]
          |([arithmetic exp])
[adding operator]:=+|-
[multiplying operator]:=*|/
```

```
[boolean exp]:=[label identifier-1]:[boolean term]{+[boolean term]}0
[boolean term]:=[boolean term]{*[boolean factor]}0
[boolean factor]:=([arithmetic exp][relational operator]
                  [arithmetic exp])|([logical exp])
[relational operator]:==|/=|>=|<=|>|<
[logical exp]:=[storage identifier-2]:[logical value-1]|
               [facility identifier-2]:[logical value-2]|
               [semaphore identifier-2]:[logical value-3]
[logical value-1]:=EMPTY|NEMPTY|FULL|NFULL
[logical value-2]:=USED|NUSED|FULL|NFULL
[logical value-3]:=ON|OFF
```

Appendix 2 Syntax of SDES Language

```
[evaluation text]:= {[PL/I st]|{[inserted st]}0}1
[inserted st]:= ${[label]:}10[traverser st];
[traverser st]:= [unstructured st]|[structured st]
[unstructured st]:= [path st]|[interaction st]|
                    [utilization st]
[structured st]:= [case st]
[path st]:= [generate st]|[terminate st]|[split st]|[assemble
           st]|[merge st]|[transfer st]|[arrive st]|[save st]|
           [regenerate st]|[save file st]|[regenerate file st]|
[interaction st]:= [attach st]|[detach st]|[lock st]|[unlock st]|
                   [trace st]|[check st]|[test st]
[utilization st]:= [seize st]|[release st]|[enter st]|[leave st]|
                  [queue st]|[depart st]|[mark st]|[tabulate st]|
                  [count st]
[generate st]:= GENERATE[traverser id]
[terminate st]:= TERMINATE[traverser id list-1]
[split st]:= SPLIT[traverser id list-1] [transfer st]
[assemble st]:= ASSEMBLE[traverser id list-1][integer]
[merge st]:= MERGER[traverser id list-1]
[transfer st]:= TRANSFER[traverser id list-1][label]
[arrive st]:= ARRIVE [traverser id list-1]|[traverser id list-3]
[attach st]:= ATTACH[traverser id list-2][PL/I expression]
[detach st]:= DETACH [traverser id list-1]|[traverser id list-2]
[save st]:= SAVE [traverser id list-1] [unique id]
[regenerate st]:= REGENERATE  [traverser id list-1] [unique id]
[save file st]:= SAVEF [traverser id list-1] [PL/I variable]
[regenerate file st]:= REGENERATF [traverser id list-1] [PL/I
                       variable]
[lock st]:= LOCK [traverser id list-1] [PL/I variable]
[unlock st]:= UNLOCK [traverser id list-1] [PL/I variable]
[trace st]:= TRACE[traverser id list-1]{[path description]}0
[check st]:= CHECK[traveser id list-1]
[test st]:= TEST{[test condition-1]|[test condition-2]}11
```

```
[seize st]:= SEIZE[traverser id list-1][facility]
[release st]:= RELEASE[traverser id list-1][facility]
[enter st]:= ENTER[traverser id list-1][storage]
[leave st]:= LEAVE[traverser id list-1][storage]
[queue st]:= QUEUE[traverser id list-1][queue]
[depart st]:=DEPART[traverser id list-1][queue]
[mark st]:= MARK[traverser id list-1][table]
[tabulate st]:= TABULATE[traverser id list-1][table]
[count st]:= COUNT[traverser id list-1]
[case st]:= CASE[PL/I variable]{[transfer st]|[generate st]}1
[traverser id list-1]:= <{[traverser id]}1>
[traverser id list-2]:= <{[traverser id].[integer]}1}>
[traverser id list-3]:= <{[traverser id]FROM[label]}1>
[path description]:= {[label]}1
[test condition-1]:= [relational op][traverser id].[integer]
                     [traverser id].[integer]
[test condition-2]:= [relational op][traverser id].[integer]
                     [PL/I expression]
[relational op]:= EQ|NE|GT|GE|LT|LE
[unique id]:= [PL/I arithmetic expression]
```